Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer[*]

Matthieu Journault[1], Antoine Miné[1,2], Raphaël Monat[1], and Abdelraouf Ouadjaout[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
`firstname.lastname@lip6.fr`
[2] Institut Universitaire de France, F-75005, Paris, France

Abstract. We discuss the design of MOPSA, an ongoing effort to design a novel semantic static analyzer by abstract interpretation. MOPSA strives to achieve a high degree of modularity and extensibility by considering value abstractions for numeric, pointer, objects, arrays, etc. as well as syntax-driven iterators and control-flow abstractions uniformly as domain modules, which offer a unified signature and loose coupling, so that they can be combined and reused at will. Moreover, domains can dynamically rewrite expressions, which simplifies the design of relational abstractions, encourages a design based on layered semantics, and enables domain reuse across different analyses and different languages. We present preliminary applications of MOPSA analyzing simple programs in subsets of the C and Python programming languages, checking them for run-time errors and uncaught exceptions.

Keywords: Static analysis · Program verification · Abstract interpretation · Tool design

1 Introduction

Static analysis aims at inferring automatically the behavior of programs in order to prove correctness properties. Abstract interpretation [4], a theory of the approximation of program semantics, helps in designing semantic-based static analyses with formal guarantees: they are sound, in that every property proved by the analyzer indeed holds; but incomplete, in that not all true properties of the program are inferred (due to incompleteness, it may fail to establish that a correct program is correct). One view we hold here, is that an *abstract interpreter* is an interpreter in the usual sense of a program interpreter computing some output and defined by induction on language syntax, except that:

1. it computes a *collecting semantics*, that collects all possible program executions along all execution paths, for all possible inputs;

[*] This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 — MOPSA.

```c
int main(int argc, char *argv[]) {
  int i = 0;
  for (char **p = argv; *p; p++) {
  l: printf("%s\n", strlen(*p)); // valid string
     i++; // no overflow
  }
  return 0;
}
```

Fig. 1. Example C program analyzed by Mopsa

Numeric:

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

Pointers:

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

Memory:

variables: `argc`, `argv`, `p`, `i`
summary block: $@$

Strings:

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

Fig. 2. Invariants inferred at label `l` for the program of Fig. 1

2. at an *abstract level*, that forgets semantic details and performs simplifications to achieve an efficient computation in a compact machine representation — a classic example is keeping variable bounds, forgetting which values are reachable within these bounds and any relationship between variable values.

An attractive feature of abstract interpretation is the existence of a variety of such abstract domains of interpretation, which target different kinds of properties and various trade-offs between cost, precision, and expressiveness. Abstract interpretation has led in the last two decades to several static analysis tools used in industry: PolySpace, Astrée [13], Sparrow [19], Julia [20], Frama-C [9], Infer [3], etc. We present here our work in progress designing Mopsa [17], a Modular Open Platform for Static Analysis programmed in OCaml. Mopsa differs from existing tools by its highly extensible, modular design, which allows easily defining and combining heterogeneous abstractions, and reusing them to analyze widely different programming languages, such as C and Python.

As a simple example, consider the small C program in Fig. 1, that prints the length of its command-line arguments, a NULL-terminated array of 0-terminated strings. Mopsa is able to prove that the string manipulation does not cause any dereference error and there is no arithmetic overflow. This is established by a combination of collaborating abstractions, as illustrated in Fig. 2: a memory abstraction partitions the memory into variables (`argc`, `argv`, . . .) and summary blocks (@) representing possibly unbounded collections of dynamically allocated blocks (here, the command-line arguments pointed to by the elements of the `argv` array); a pointer abstraction maintains points-to information ($\texttt{p} \mapsto \{\texttt{argv}\}$); a string abstraction maintains predicates on the position of the terminating 0 ($\exists k \in [0 \dots \texttt{size}(@) - 1] : @[k] = 0$); and a numeric abstraction infers ranges and affine inequalities. Despite abstracting very different objects, these domains obey a common signature and are loosely coupled, and so can be easily plugged in and out. Moreover, they collaborate in several ways:

1. Cartesian products, to combine domains discussing about orthogonal semantic objects (such as pointer variables and numeric variables);
2. reduced products, to combine domains abstracting the same semantic object in different ways (such as interval and polyhedra [8]);
3. delegation, for a domain to rely on another one for its computations (*e.g.* the pointer domain relies on numeric domains to maintain offset information).

When combined, these mechanisms allow a powerful interaction between domains. For instance, Fig. 2 shows that it is possible to infer affine relations between integer variables and other integer quantities introduced by the other abstract domains, such as pointer offsets and string sizes.

Section 2 presents our representation of programs in Mopsa, which is close to the source level to avoid losing high-level information and uses extensible abstract syntax trees to support the addition of analysis targets. This is in contrast to traditional analyzers, that translate source programs into a simplified, fixed, low-level representation (such as simplified C, or LLVM bitcode), on which the semantic analysis is performed. Section 3 presents the dynamic simplification of expressions performed during the analysis. This again contrasts to traditional analyzers, where front-ends perform static simplifications, and it is key to achieve a flexible delegation mechanism of domain computations while keeping a fully relational analysis. Section 4 details how the combination of domains is achieved. Section 5 presents our application to analyzing a large subset of the C and Python languages, as well as preliminary experimental results. Although Mopsa is not yet able to analyze large-scale C nor realistic Python programs, we believe that these results are encouraging as few tools have yet shown the ability to analyze languages as dynamic as Python, nor been able to factor the analysis of such different languages as C and Python in the same framework. Moreover, Mopsa is intended as a platform for research, and has been exploited in several exploratory works on analyzing Python [10], strings [11], and trees [12]. We plan to release our implementation as open-source software. Section 6 concludes. This article extends on a short presentation of Mopsa from [17] by giving notably more details and examples on Mopsa's abstractions, dynamic expression simplification, domain compositions, as well as more recent benchmarks.

2 Unified Extensible Language

Classic static analyzers operate on an intermediate language — such as LLVM bitcode [14] — rather than the source language. One benefit is that the semantic analysis needs to handle far less constructions, with a simpler semantics. Supporting a new target language is then only a matter of writing a new front-end that translates it into this fixed intermediate language. However, some information is lost in the translation, which may hurt the subsequent analysis (*e.g.* LLVM forgets whether integer types are signed or unsigned, while transformation to 3-address code puts a strain on relational domains to maintain precision [18]). Additionally, a common intermediate language may not fit all possible language kinds. On the contrary, MOPSA employs an extensible AST data-type to keep as much high-level information as possible and be open for new targets. Each analyzer module can define additional variants for syntactic objects: statements, expressions, types, variables, etc. Currently, MOPSA supports the following:

– Universal, a toy-language that mainly features an unbounded integer datatype and simple control constructs (loops, conditionals, functions);
– most of C, through Clang's parser, and a contract annotation language inspired by ACSL [9] to model library functions (Sect. 5.2);
– a large subset of Python 3 (using a dedicated parser).

Extensible Syntax. Using OCaml's extensible variant types, any OCaml module can extend MOPSA's AST. For instance, MOPSA's abstract syntax for simple `while` loops in Universal is introduced as:

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

Then, the C syntax module defines loops as:

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

We do not redefine `while` loops for C as they are identical to the ones in Universal. However, we add `for` and `do-while` loops, which have a different syntax, so as to keep separate each kind of loops in the program representation, instead of lowering them to `while` loops. For completeness, Python loops are defined as:

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

They feature an additional statement, which denotes the else clause of the loops.

Distributed Iterator. The semantic effect is defined by a domain which provides an `exec` function. Like the syntax, the `exec` function can be distributed among several modules. A global iterator will call in turn the `exec` functions until one of them returns a non-`None` value. For instance, the semantics of `while` loops in Universal is defined as a fixpoint as follows, where `flow` represents the flow of abstract information, `lfp` performs classic fixpoint iteration with acceleration, and `join` computes the union of abstract information:

```
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
   in Some (man.exec (S_assume (E_not cond) i))
| _ ->
   None (* pass-through to the next domain *)
```

The semantics of loops is defined in terms of the semantics of the loop body and conditions (`S_assume`), hence, its `exec` function must be able to call the global iterator that will in turn find the proper module to handle these statements; this is the role of the *manager* `man` passed as argument to all abstract functions.

The semantics of a C `for` loop can be defined in terms of Universal loops as a simple syntactic transformation, which will be automatically delegated to Universal's loop iterator, after executing the loop initialization statement:

```
let exec stmt man flow = match stmt_kind stmt with
| S_c_for (init, cond, incr, body) ->
   let body' = ... in
   Some (man.exec (mk_block [init; S_while (cond, body')]) flow)
| _ -> None
```

This has the benefit of factoring the logic for fixpoint computation in one place. Such a translation is done at analysis time: it could access information from the abstract state (through `flow`) and manipulate it (through `man`). While this is not the case for this simple example, we will see its benefit in Sect. 3.

Domains. Traditional analyzers separate iterators, that work on high-level control structures or control-flow-graphs, from abstract domains, that ultimately handle atomic statements, such as assignments and tests. However, Mopsa uses the same signature for both. The same way the global `exec` function available through the manager `man` is composed of `exec` functions defined in all domains, the abstract flow of information `flow` is composed of information from each domain (such as intervals, pointers, etc.) and can be manipulated by any domain (*e.g.* to perform joins). A loop iterator happens to be a domain with empty abstract state, and the policy of finding the first domain that handles a specific statement is one example of domain composition, called sequence. Other domain composition operators, such as reduced and Cartesian products, are discussed in Sect. 4. A typical analysis instance contains dozens of domains. This is illustrated in Fig. 3 for the case of the C analyzer (we give more details on these domains in Sects. 4 and 5). Note that many domains defined for Universal are reused in the C analysis.

3 Dynamic Expression Rewriting

When analyzing a program, the original AST goes through successive transformations that reduce its complexity. For instance, an assignment in C can be first translated into a simplified subset of C by flattening all data structures

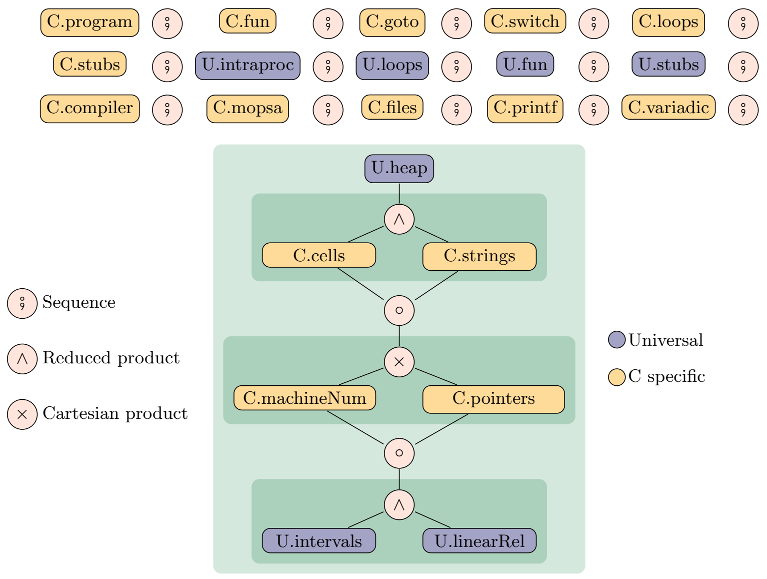

Fig. 3. Domain composition in Mopsa's C analyzer (see also Sect. 5.1)

into arrays, before moving to another subset containing only scalar variables by resolving dereferences. These transformations are performed dynamically during the analysis in order to gain in precision by leveraging the inferred constraints.

Evaluating to Expressions. An important particularity of these translations in Mopsa is the evaluation mechanism of expressions. In order to preserve relational constraints present in the original structure of the program, expressions are not evaluated into abstract values, but into other expressions.

As an illustration, consider the assignment `x = a[i] + 1` in the Universal language. The numeric domain is responsible for assignments to integer variables, but can not handle expressions with arrays. Therefore, it delegates, with `eval`, the evaluation of the right-hand-side expression to other domains via the manager:

```
let exec stmt man flow = match skind stmt with
  | S_assign(E_var x, e) ->
    let e' = man.eval e flow in
    ...
  | _ -> None
```

A domain that abstracts arrays as smashed variables implements the evaluation of `a[i]` by providing an `eval` function returning an auxiliary numeric variable, `a*`, introduced by the abstraction to denote the values of all the array elements:

```
let eval exp man flow = match ekind e with
  | E_subscript(E_var a,i) ->
```

```
    let smash = mk_smash a in
    Some (Eval.return smash flow)
  | _ -> None
```

The original statement `x = a[i] + 1` is thus translated into the simplified assignment `x = a* + 1`, which can be handled directly by the numeric domain.

Keeping the right-hand side of the assignment symbolic allows a relational domain to infer relations between program variables. Consider, for instance, a memory domain abstracting arrays by expansion, creating variables `a0`, `a1`, ... to denote the values of `a[0]`, `a[1]`, ... Then, in an environment where $\mathtt{i} = 1$, the domain will translate `x = a[i] + 1` into `x = a1 + 1`, allowing a domain such as polyhedra [8] to maintain the relation $\mathtt{x} = \mathtt{a1} + 1$.

Note that neither adding an array data-type, nor choosing whether to abstract by smashing or by expansion, required any change to the numeric domain; the new abstraction is conveyed through expression rewriting. This mechanism makes it easy to reuse existing abstractions with novel operators and semantics. As another example, while Universal features arithmetic on mathematical integers, which fits classic numeric domains well, C expressions are evaluated using machine integers. Thus, we added a domain that translates machine arithmetic into integer arithmetic. It checks for overflows in the current abstract state, and exploits the fact that, in the absence of overflow, the two semantics coincide, to output, when possible, expressions that are close to the original ones.

Disjunctions. Evaluations in Mopsa also offer an elegant way to perform a case analysis. Domains can evaluate an expression into a disjunction of different expressions, for different subsets of the abstract state. To manipulate these disjunctive evaluations easily, two mechanisms are provided. Firstly, domains can use a monadic bind operator `>>=` to execute a transfer function on each case of the evaluation. Secondly, abstract versions of common test statements, such as `if` and `switch`, are introduced to express guarded evaluations.

Consider an abstraction of 0-terminated C strings [11] that abstracts strings with their length, *i.e.* the position of the first 0 in the array. To each memory block `b`, the domain associates an integer auxiliary variable $\mathtt{b}_l$ such that:

$$\mathtt{b}[\mathtt{b}_l] = 0 \wedge \forall i \in [0, \mathtt{b}_l - 1] : \mathtt{b}[\mathtt{b}_l] \neq 0$$

Using this auxiliary variable, the evaluation of an access `b[i]` is decomposed into three cases using the abstract operator `switch`. Indeed, depending on the ordering between `i` and $\mathtt{b}_l$, `b[i]` may evaluate to a non-null, a null, or an arbitrary byte value. Each case is a pair containing a guard and the associated expression:

```
let eval exp man flow = match ekind e with
| E_c_subscript(b, i) ->
  (* Evaluate the index expression *)
  man.eval i flow >>= fun i' flow ->
  ...
  (* After checking out-of-bound accesses *)
  let length = mk_length b in
  Some (switch [
    (* Case 1: access before the first 0 *)
    (mk_lt i' length), (* i < length(b) *)
    (fun flow -> Eval.return (mk_interval 1 255) flow); (* [1,255] *)

    (* Case 2: access at the first zero *)
    (mk_eq i' length), (* i = length(b) *)
    (fun flow -> Eval.return zero flow); (* 0 *)

    (* Case 3: access after the first zero *)
    (mk_gt i' length), (* i > length(b) *)
    (fun flow -> Eval.singleton (mk_interval 0 255) flow) (* [0,255] *)
  ] man flow)

| _ -> None
```

The conditions $\mathtt{i} < \mathtt{b}_l$, $\mathtt{i} = \mathtt{b}_l$ and $\mathtt{i} > \mathtt{b}_l$ are interpreted by numeric domains and refine the abstract environments during the evaluation. Then, an assignment such as `x = b[i]` will trigger three assignments in the numeric domain, one for each case, after which the cases are merged with an abstract join. When using a relational numeric domain, this allows us to infer easily relations between auxiliary variables, such as $\mathtt{b}_l$, and program variables.

4 Domain Combination

Mopsa provides several combination operators that simplify the construction of complex abstractions, such as the Cartesian product operator $\textcircled{\times}$, the sequence operator $\textcircled{;}$, and the reduced product operator $\textcircled{\wedge}$. To preserve modularity, domains should be loosely coupled by keeping their abstraction private. On the other hand, domains are not isolated and need to cooperate in order to exploit the available abstractions.

Queries. Similarly to input channels in Astrée [6], domains in Mopsa can request the computation of abstract properties, such as the interval of a numeric expression. This is done by defining a *query*, which is an extensible GADT type encoding the query argument and its result. For instance, the interval query can be defined as follows:

```
type _ query += Q_interval: expr -> (int option * int option) query
```

Answering to queries is done by defining a transfer function `ask` in the domain:

```
let ask : type r. r query -> t -> r option = fun query state ->
  match query with
  | Q_interval e ->
    let l, u = ... in
    Some (l, u)
  | _ -> None
```

A domain returns `None` for queries it cannot handle. Client domains can retrieve this information via their manager by calling `man.ask (Q_interval e) flow`, and do not need to know which domain(s) can answer. Queries come with lattice operators to combine the replies from several domains.

Reductions. Reduced products [5,6] are a common example of cooperation in abstract interpreters. A reduced product computes the intersection of domains approximating the same concrete semantics, and allows refining the abstract state of a domain by exploiting information computed by the other domains.

To illustrate this form of cooperation, consider the classic example of reducing intervals and congruences [16]. Given an interval $[11, 12]$ and a congruence $2\mathbb{Z}+1$, we can refine both values in two steps. Firstly, using the fact that the value is odd, the interval is refined into $[11, 11]$. After that, since the interval is now a singleton, the congruence is refined into $0\mathbb{Z} + 11$.

Defining reductions in Mopsa is different than in existing analyzers in several ways. Firstly, it is simpler while being powerful enough to define complex reductions. Reduction rules are not part of the transfer functions of domains and do not require using particular communication channels to retrieve required information. Instead, they are defined externally in separate modules with a simplified signature that allows access to the internal representation of abstract elements easily. It is thus easy to design new reductions, or remove them, while keeping the core transfer functions of the abstract domains unchanged. For instance, the reduction between intervals and congruences is defined as:

```
let reduce man pointwise =
  let i = man.get Interval.id pointwise
  and c = man.get Congruence.id pointwise in
  let i', c' = meet_interval_congruence i c in
  man.set Interval.id i' pointwise |>
  man.set Congruence.id c'
```

Secondly, in contrast to Astrée [6], there is no fixed order of computation in a reduced product. Post-conditions are computed independently, before applying reduction rules on the pointwise result. Finally, reduced products in Mopsa are not limited to iterated pairwise reductions, but support n-ary reduction rules, which enables more precision [7].

Sharing. While many abstract interpreters offer the possibility to build reduced products [2,9], a distinctive feature of Mopsa is the ability to define products of abstract domains that share a part of their abstraction. For instance, the C analysis (Fig. 3) features a Cartesian product of a domain `C.machineNum` handling statements over C numeric expressions by rewriting them into mathematical integer expressions, and a domain `C.pointers` handling C pointer expressions and storing points-to information. These domains are assembled in a Cartesian product as their semantics do not overlap: unlike reduced products, they target orthogonal expressions. However, both abstractions delegate a part of their state to an underlying numeric domain: integer C variables for `C.machineNum`, and pointer offsets for `C.pointers`. Mopsa offers the possibility for these two domains to share some underlying abstract state (denoted as ⊙ in Fig. 3). We can thus exploit a numeric domain to infer relations between pointer offsets and integer variables. As a consequence of this sharing, the composition of abstract states from individual domains forms a DAG, not a tree.

Logs. As for the Cartesian product, Mopsa allows domains composed in a reduced product to delegate part of their abstraction to another (potentially shared) domain. This is the case for the reduced product between the `C.cells` and the `C.strings` domains in the C analysis of Fig. 3. The cell domain [15] is used to represent the C memory: it translates the semantics of all C memory accesses into a semantics over a set of scalar variables. Abstracting the values of these variables is then delegated to an underlying abstraction. Recall that the string domain represents the length of a C string, *i.e.* the position of the first 0, by associating a numeric variable to each memory region [11]. The cell and string domains provide information on common parts of the C memory, hence, they are composed in a reduced product. Sharing the underlying domain allows the discovery of relations between string lengths (managed by the string abstraction), and numeric and pointer variables (managed by the cell abstraction).

In the case of a Cartesian product, a statement is always handled by at most one of the domains in the product. In contrast, in the case of a reduced product, the statements are handled by all the domains. Each domain transforms the shared underlying domain, inducing several different states for the shared component, which must then be merged into a sound post-condition.

As an example, consider a simplified abstraction defined as the reduced product between the cell and string domains delegating to a shared numeric domain. Consider moreover the statement $s \overset{\text{def}}{=}$ `a[0] = '\0'` executed in the shared numeric abstract state $S^\sharp = \{\mathtt{a}_0 = 1, \mathtt{a}_l \geq 3\}$, where $\mathtt{a}_l$ is the variable encoding the length of string `a` (managed by the string domain) and $\mathtt{a}_0$ is the variable representing the values of `a[0]`, the first character of the `a` string (managed by the cell domain). The cell abstraction will translate s into $\mathtt{a}_0$ `= 0`. The string abstraction will translate s into $\mathtt{a}_l$ `= 0` (indeed the length of string `a` will be 0). The execution of the abstract statements on $S^\sharp$ yields the two following abstract states: $S_1^\sharp = \{\mathtt{a}_0 = 0, \mathtt{a}_l \geq 3\}$ and $S_2^\sharp = \{\mathtt{a}_0 = 1, \mathtt{a}_l = 0\}$. Neither state is a sound post-condition for statement s. Indeed, the effect of the statement should update both variables, but here each domain instead only updates its own variable. $S_1^\sharp$ and $S_2^\sharp$ must therefore be *merged* into a sound post-condition. In our example, the abstract elements can be merged by forgetting the constraints on the variables modified by the other domain, and then intersecting the two resulting abstract states, yielding $S_r^\sharp = \{\mathtt{a}_0 = 0, \mathtt{a}_l = 0\}$, which is a sound post-condition containing both the transformations induced by the cell and string domains. In order to know which variables were modified by the other component of the product, we automatically log the list of statements that were applied on each abstract element, and use these two logs to merge together the parts of the abstraction that we want shared.

To sum up, the computation of post-conditions is done independently on each abstract domain as if no sharing was present, then the tree of domains is merged back into a DAG. The process is mostly automated, and does not require any action from the domains in the reduced product. It is sufficient that all shareable domains, such as numeric domains, define a suitable merging function.

5 Implementation and Applications

Mopsa is written in OCaml. Parsers and utilities account for 19,000 lines of code (we used the `cloc` command to measure the length of our files). The framework, describing the structure of abstract domains and the domain combinators, consists in 8,000 LOC. The analysis of Universal takes 3,000 LOC, while the one of C and its stubs takes 7,100 LOC, and the analysis of Python is 7,700 LOC long.

5.1 C Analysis

Mopsa performs a reachability analysis of C programs to infer invariants and report run-time errors, such as arithmetic overflows, invalid pointer uses, or failed assertions. Mopsa first parses the source files using a front-end based on Clang, and converts the AST to OCaml, keeping all C high-level syntax and type information. The files are then linked, *i.e.* merged into a single AST by resolving symbol definitions. The analyzer is then called on the `main` entry-point using the configuration of abstract domains currently described in Fig. 3. This configuration is naturally intended to evolve as new abstractions are introduced.

Iterators. The configuration starts with a long sequence of iterators, `C.program` to `C.variadic`, *i.e.* state-less domains that handle individual parts of the C compound syntax by induction, including loops, `switch`, `goto`, etc. As explained in Sect. 2, the configuration merges domains reused from the Universal toy-language and C-specific domains. C domains often delegate to Universal ones, *e.g.*, in the case of loops (respectively `C.loops` and `U.loops`). As another example, Mopsa currently handles function calls by semantic inlining (*i.e.* calling recursively the iterator on the function body at each call), which is implemented in a Universal domain `U.fun` (although we are experimenting with summary-based modular function analyses [11]). A C-specific domain, `C.fun`, translates C function calls into Universal ones, taking care of C-specific aspects such as calls through function pointers. Additional domains handle special calls, such as variadic arguments (`C.variadic`), calls to builtin analyzer functions (such as `printf` or file operations) or user-defined stubs (Sect. 5.2).

Domains. Following these iterators, the C analysis contains a composition of domains that handle atomic statements such as assignments and tests. Dynamic memory is handled by `U.heap` using recency abstraction [1]: each allocation site is associated with at most two abstract blocks, one representing the lastly allocated block at this site (on which we can perform strong updates, which is critical for precision), and one representing all the allocated blocks before (on which we must perform weak updates) — this domain could be easily replaced with any domain that partitions the possibly unbounded set of allocated blocks into a bounded set of abstract blocks. Each variable or abstract heap block is then decomposed into a set of virtual variables, called *cells*, of scalar type, by `C.cells`. In order to handle transparently union types and type-punning, we use the cell abstraction from [15], where the decomposition is adapted dynamically according to the actual access pattern during the execution (rather than based on the static type, which can be deceiving). As explained in Sect. 4, the cell domain is composed using a reduced product with a string abstraction, `C.strings`, tracking the position of 0 in character arrays. Both domains are able to rewrite expressions into dereference-free expressions on scalar variables. These are handled by a Cartesian product: `C.machineNum` translates machine integer arithmetic into mathematical arithmetic, handing overflow-checking and wrap-around semantics; while `C.pointers` translates pointer arithmetic into byte-offset arithmetic while maintaining in its internal abstract state the bases (*i.e.* pointed-to variables) of each pointer. Both these domains collaborate to rewrite scalar expressions into expressions on mathematical integers, which are then handled natively by classic numeric abstract domains, such as integer intervals (`U.intervals`). Mopsa also features a rational polyhedra domain (`U.linearRel`), which is a work in progress and was not enabled in our benchmarks. Floating-point arithmetic is also supported, using intervals, but not shown here for simplicity.

Benchmarks. To assess the effectiveness of Mopsa, we have analyzed real-world C programs from the GNU Coreutils package, which is a collection of command-line utilities. Mopsa was easily integrated into the `make`-based build system, without modifying any source file or build script. We have also tested Mopsa on a part of the Juliet test suite developed by NIST. These programs differ from Coreutils as they are composed of a large number of small functions for testing analyzers on common software weaknesses.

The results are summarized in Table 1. As the analyzer is still a work in progress, not all programs from the benchmarks were analyzed. For Coreutils programs, each analysis terminated under 10s, while the number of reported alarms was generally high. This imprecision is mainly due to the absence of an adequate abstraction of pointer arrays, which is currently under development. For Juliet, we focused on three kinds of weaknesses relevant to Mopsa: NULL-pointers, integer overflows, and divisions by zero. The results for Juliet tests were more precise, as they do not employ complex data structures. Nevertheless, the analysis of the CWE369 tests for assessing the detection of divisions by zero shows a high imprecision rate due to the absence of a partitioning abstraction.

Table 1. Benchmark results for the analysis of some programs from GNU Coreutils v8.30 and NIST Juliet v1.3

Benchmark	Name	LOC	Time	Alarms
GNU Coreutils v8.30	true	759	6.92s	20
	printenv	814	6.66s	20
	getlimits	970	8.83s	119
	test	1,238	7.06s	0
	runcon	1,295	7.16s	11
	comm	2,634	7.63s	46
	hostid	2,730	7.61s	31
	id	2,733	7.12s	31
	logname	2,735	7.58s	32
	whoami	2,742	4.69s	36
	link	2,747	7.76s	42
	nice	2,955	6.92s	24
	sleep	3,151	7.87s	35
NIST Juliet v1.3	CWE476	25k	5min51s	0
	CWE369	109k	17min00s	324
	CWE190	440k	1h22min56s	0

5.2 C Stub Modeling

To soundly analyze a C program, Mopsa needs to know the semantic effect of every function that can be called, directly or indirectly, from the `main` entry-point, including all library functions. Linking the full source of all libraries is not always possible (low-level functions may be written in assembly or use compiler intrinsics) nor convenient (as they may be large). Solutions include hard-coding in the analyzer the effect of these functions, or writing a *stub*, *i.e.* a C function modeling its effect as done in Astrée [13] and Frama-C [9]. Instead, Mopsa introduces a dedicated modeling language to ease the quick specification of stubs, and ensure a fast and precise analysis. This language is inspired by ACSL: Frama-C's contract language [9] using pre/post-condition directives put in special comments at function declarations. For instance, the specification of `strlen` is:

```
/*$
 * requires: valid_string(s);
 * ensures: return in [0, size(s)-1];
 * ensures: s[return] == 0;
 * ensures: forall unsigned int k in [0, return-1]: s[k] != 0;
 */
size_t strlen (const char *s);
```

where the predicate `valid_string` is defined as:

```
/*$$
 * predicate valid_string(s):
 *   valid_ptr(s) and
 *   exists int i in [0, size(s)-1]: s[i] == 0;
 */
```

Whenever encountering a call to such a specification, Mopsa checks that the pre-condition (here, that `s` points to a valid string) is satisfied, and reports a run-time error if this is not true.

In Mopsa, the modeling language does not benefit from a special treatment: it is simply another language that extends the global AST with its own syntax, including logic connectors, such as `forall` or `and`, built-in functions, such as `valid_ptr` and `size`, while reusing the syntax of side-effect free C expressions. Whenever a function with a model available is called, this model AST is interpreted. This is handled by the iterators `C.stubs` and `U.stubs` from Fig. 3, which interpret contracts by relying on a translation into assertions (to verify pre-conditions) and assignments (to enforce post-conditions) of C expressions, using the same mechanism as described in Sect. 2. Additionally, the string domain has been enriched to interpret the simple logical expressions found in the model of `strlen` and `valid_string`: $\forall k \in [0, r] : s[k] \neq 0$ and $\exists k \in [0, r] : s[k] = 0$. The domain is actually able to analyze either the model of `strlen` or an actual implementation of `strlen` with the same degree of precision.

While similar to logical languages used in deductive methods, such as the WP plugin of Frama-C [9], our modeling language is used in quite a different way. Firstly, it is not used to check the implementation of a function with respect to a functional specification, but rather to replace a function having no implementation with this specification. Secondly, while deductive methods rely on powerful, but costly automated theorem provers to check expressive classes of quantified logic formulas, we rely instead on fast abstract domains that are generally only able to process a very restricted subset of logic formulas, with very specific shapes, but do so in a way consistent with the abstract information they encode — for instance, the string domain only matches simple formulas stating the presence or absence of a 0 in a partition of an array. An interesting point is that C and stub modeling employ quite different kinds of languages, respectively an imperative and a logical language. Mopsa thus achieves a form of multi-lingual analysis, analyzing mixed programs by combining abstractions dedicated to each language while sharing common abstractions.

5.3 Python Analysis

Python's configuration (shown in Fig. 4) is currently simpler than its C counterpart, as it focuses on finding type errors rather than low-level numeric properties. Nevertheless, as Python is a very dynamic language, finding statically such type errors is both difficult and useful for programmers. We also plan to add a value analysis in Mopsa [10]. Python's configuration is composed of several parts:

– `Py.program`, which takes care of program parsing.
– A desugarization, focusing on translating Python-specific control-statements into the Universal language.
– `Py.exceptions`, handling the analysis of specific control-flow statements such as exceptions.
– Universal iterators, handling the intraprocedural (`U.intraproc`, `U.loops`) and interprocedural (`U.fun`) analysis of statements in the Universal language.

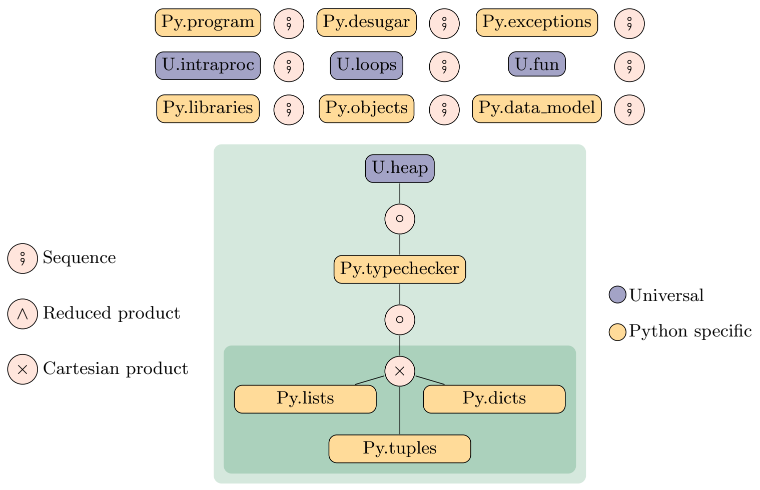

Fig. 4. Domain composition in Mopsa's Python analyzer

Table 2. Analysis of official Python benchmarks

Name	LOC	Analysis time	# Alarms	# False Alarms
bm_fannkuch.py	59	0.07s	0	0
bm_float.py	63	0.06s	0	0
bm_spectral_norm.py	74	0.33s	0	1
bm_nbody.py	157	1.5s	0	1
bm_chaos.py	324	5.6s	1	0
bm_unpack_sequence.py	458	3.1s	0	0
bm_hexiom.py	674	2m58s	0	52

- A domain implementing the abstract effect for some parts of Python's vast standard library (this modeling is currently hard-coded in OCaml and does not use a modeling language as for C).
- A domain handling Python objects such as classes and functions.
- A description of Python's data model, encoding the semantics of built-in Python operators, such as attribute accesses, arithmetic operations and subscript operators.
- The stateful part of the analysis, composed of the recency abstraction `U.heap` [1] from Universal; an abstract domain of Python types; and, finally, a smashing-based abstraction of data containers, such as lists and dictionaries, while tuples are abstracted by expansion. Note that the data container abstraction is defined independently from the type analysis, and could be reused in a value analysis instead.

We show the results of our analysis in Table 2, on benchmarks used by the standard Python interpreter.[3] We focused on 7 benchmarks, which were chosen for their low number of external dependencies. We found one `TypeError` in `bm_chaos.py`,[4] which was never reached in the actual test, but could be triggered by instantiating a class using non-default arguments. The last benchmark `bm_hexiom.py` has a number of false alarms due to our analysis being unable to distinguish empty lists from non-empty ones.

6 Conclusion

We presented the design of our platform for static analysis by abstract interpretation, based on the idea of a collaboration of loosely coupled, highly reusable abstractions. Compared to existing analysis platforms, it makes a few original choices: using a unified extensible abstract syntax tree to both represent faithfully high-level source languages as well as intermediate languages, unifying iterators and abstract domains, domain collaboration through dynamic expression rewriting, as well as reduced and Cartesian products, possibly sharing abstract state. Currently, our OCaml implementation is not yet able to analyze large programs. Yet, we demonstrated the feasibility of our approach by implementing abstractions of different kinds (numeric as well as pointer, dynamically allocated memory, structured types, objects, strings, etc.) and applying them to the reachability analysis of two widely different languages: Python and C (including a library modeling language). In addition to the implementation of classic abstractions, Mopsa has also been used to implement and test novel abstract domains such as [10], [11], and [12].

Future work will include improving our implementation to reliably analyze realistic C and Python programs, notably with a better support for libraries. We will also consider incorporating novel abstractions into our framework to improve precision and efficiency, or to prove properties beyond the absence of run-time errors. The framework could also be extended to support backward analysis and incremental analysis. Finally, we will consider supporting new target languages. An interesting aspect is that syntax and abstractions targeting different languages can be included in an analyzer configuration, which opens the possibility of analyzing a program written in several languages. We had some success combining C programs with libraries modeled in a logic-based contract language. It would be an interesting challenge to consider the analysis of programs mixing C and Python.

References

1. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Static Analysis: 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings. pp. 221–239. Springer (2006)

[3] `https://github.com/python/pyperformance/`

[4] `https://github.com/python/pyperformance/issues/57`

2. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace. pp. 1–38. No. 2010-3385, AIAA (Apr 2010)
3. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NFM. pp. 3–11. Springer (2015)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77. pp. 238–252. ACM (Jan 1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of POPL'79. pp. 269–282. ACM Press (1979)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: Proc. of ASIAN'06. LNCS, vol. 4435, pp. 272–300. Springer (Dec 2006)
7. Cousot, P., Cousot, R., Mauborgne, L.: The reduced product of abstract domains and the combination of decision procedures. In: Proc. of FoSSaCS '11, LNCS, vol. 6604, pp. 456–472. Springer-Verlag (2011)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78). pp. 84–97. ACM (1978)
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**, 573–609 (2012)
10. Fromherz, A., Ouadjaout, A., Miné, A.: Static value analysis of Python programs by abstract interpretation. In: Proc. of NFM'18. pp. 185–202. LNCS, Springer (Apr 2018)
11. Journault, M., Ouadjaout, A., Miné, A.: Modular static analysis of string manipulations in C programs. In: Proc. of SAS'18. LNCS (2018)
12. Journault, M., Miné, A., Ouadjaout, A.: An abstract domain for trees with numeric relations. In: ESOP. Lecture Notes in Computer Science, vol. 11423, pp. 724–751. Springer (2019)
13. Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Astrée: Proving the absence of runtime errors. In: Proc. of ERTS2 2010 (May 2010)
14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. of CGO'04 (Mar 2004)
15. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of LCTES'06. pp. 54–63. ACM (Jun 2006)
16. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages (FnTPL) **4**(3–4), 120–372 (2017)
17. Miné, A., Ouadjaout, A., Journault, M.: Design of a modular platform for static analysis. In: Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18). p. 4. Lecture Notes in Computer Science (LNCS) (28 Aug 2018)
18. Namjoshi, K.S., Pavlinovic, Z.: The impact of program transformations on static program analysis. In: Static Analysis. pp. 306–325. Springer, Cham (2018)
19. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. SIGPLAN Not. **47**(6), 229–238 (Jun 2012)
20. Spoto, F.: Julia: A generic static analyser for the Java bytecode. In: Proc. of FTfJP'2005. p. 17 (July 2005)