A Library Modeling Language for the Static Analysis of C Programs

Abdelraouf Ouadjaout, Antoine Miné

A Library Modeling Language for the Static Analysis of C Programs*

Abdelraouf Ouadjaout[1] Antoine Miné[1,2]

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

Abstract. We present a specification language aiming at soundly modeling unavailable functions in a static analyzer for C by abstract interpretation. It takes inspiration from Behavioral Interface Specification Languages popular in deductive verification, notably Frama-C's ACSL, as we annotate function prototypes with pre and post-conditions expressed concisely in a first-order logic, but with key differences. Firstly, the specification aims at replacing a function implementation in a safety analysis, not verifying its functional correctness. Secondly, we do not rely on theorem provers; instead, specifications are interpreted at function calls by our abstract interpreter.
We implemented the language into Mopsa, a static analyzer designed to easily reuse abstract domains across widely different languages (such as C and Python). We show how its design helped us support a logic-based language with minimal effort. Notably, it was sufficient to add only a handful transfer functions (including very selective support for quantifiers) to achieve a sound and precise analysis. We modeled a large part of the GNU C library and C execution environment in our language, including the manipulation of unbounded strings, file descriptors, and programs with an unbounded number of symbolic command-line parameters, which allows verifying programs in a realistic setting. We report on the analysis of C programs from the Juliet benchmarks and Coreutils.

1 Introduction

Sound static analysis of real-world C programs is hampered by several difficult challenges. In this work, we address the key problem of analyzing calls to external library functions, when analyzing library code is not an option (e.g., it is unavailable, has unsupported features such as system calls or assembly). More specifically, we target the GNU implementation of the C library [13], a library used in a large number of applications and featuring thousands of functions covering various aspects, such as file management, socket communication, string processing, *etc.* Several approaches have been proposed to analyze programs that depend on such complex libraries:

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

```
1 size_t strlen(const char* s) {
2   int size;
3   __require_allocated_array(s);
4   size = __get_array_length(s);
5   return size - 1;
6 }
```

(a) Stub of `strlen` in Infer

```
1 /*@ requires: valid_read_string(s);
2   @ assigns \result \from indirect:s[0..];
3   @ ensures: \result == strlen(s);
4   @*/
5 size_t strlen (const char *s);
```

(b) Stub of `strlen` in Frama-C

```
1 /*$
2  * requires: s != NULL ∧ offset(s) ∈ [0, size(s));
3  * requires: ∃i ∈ [0, size(s)-offset(s)): s[i] == 0;
4  * ensures : return ∈ [0, size(s)-offset(s));
5  * ensures : s[return] == 0;
6  * ensures : ∀i ∈ [0, return): s[i] != 0;
7  */
8 size_t strlen(const char s);
```

(c) Stub of `strlen` in Mopsa

```
1 int n = rand()%100;
2 char *p = malloc(n + 1);
3 if (!p) exit (1);
4 for(int i=0;i<n;i++)
5   p[i] = 'x';
6 a[n] = '\0';
7 int m = strlen(p);
```

(d) Example with `strlen`

Fig. 1. Examples of stubs in different analyzers.

Stubs as C code. A common solution is to provide alternative C implementations of the library functions, called *stubs*. In order to remain sound and be effectively analyzed, stubs are generally simpler and contain calls to special builtins of the analyzer that provide more abstract information than the classic constructs of the language. This approach is adopted by many static analyzers, such as Astrée [4] and Infer [6]. For example, Fig. 1a shows the stub of `strlen` in Infer: it uses builtin functions to check that the argument points to a valid block before returning its allocation size. The approach makes it difficult for the stub programmer to express complex specifications with higher levels of abstractions, as key parts of the semantics are hidden within the builtin implementation. Moreover, writing stubs as C code and hard-coding builtins is acceptable when targeting embedded code [4], that does not rely much on libraries, but is not scalable to programs with more dependencies.

Stubs as logic formulas. More adapted specification languages have been proposed to overcome these drawbacks, principally based on formulas written in first-order logic. Some of them exploit the flexibility of the host language in order to define an *embedded domain specific language*, such as CodeContracts checker [11] that can express specifications of C# functions in C# itself. Other solutions propose a dedicated language and specifications are written as comments annotating the function. The most notable examples are JML for Java [18] and ACSL for C [3]. They have been widely used in deductive verification, employing theorem provers that naturally handle logic-based languages, but less in value static analysis by abstract interpretation. We show in Fig. 1b the specification of `strlen` in ACSL, as defined by Frama-C's value analyzer [9]. The syntax is less verbose than the C counterpart. Yet, essential parts of the stub

are still computed through builtins. It is worth noting that Frama-C features another, more natural, specification of `strlen`, exploiting the expressiveness of the logic to avoid builtins. But this specification is large (64 lines) and employs quantified formulas that are too complex for the value analysis engine: it is used only by the deductive verification engine.

Abstract interpretation of logic formulas. In this paper, we propose a novel approach based on abstract interpretation [7] that can interpret specifications written in a logic-based language of library functions when they are called. Similarly to CodeContracts checker [11], we do not rely on theorem provers to interpret formulas; instead, specifications are interpreted by abstract domains tailored to this task. The key novelty of our solution is that we consider the logic language as a separate language with its own concrete and abstract semantics, while contracts in cccheck are embedded within the host language as function calls. We believe that this decoupling makes the design more generic and the language is not limited by the semantic nor the syntax of the host language.

We implemented the proposed approach into Mopsa [16], a static analyzer that features a modular architecture that helps reusing abstract domains across different languages. We leverage this modularity and we illustrate how we can improve the analysis by extending C abstract domains to add transfer functions that exploit the expressiveness of formulas and infer better invariants. For example, the stub of `strlen` as defined in Mopsa is shown Fig. 1c. It relies essentially on constraints expressed as formulas instead of specific analyzer builtins. These formulas can be handled by Mopsa, and string lengths can be computed precisely even in the case of dynamically allocated arrays. For instance, at the end of the program shown in Fig. 1d, Mopsa can infer that $\texttt{m} = \texttt{n}$.

Contributions. In summary, we propose the following contributions:

- We present in Sec. 2 a new specification language for C functions and we formalize it with an operational concrete semantic. In addition to standard constructs found in existing languages, it features a resource management system that is general enough to model various objects, such as blocks allocated by `malloc`/`realloc` or file descriptors returned by `open`. Illustrative examples can be found in App. A.
- We present in Sec. 3 a generic abstract domain for interpreting the specification language, that is agnostic of the underlying abstraction of C.
- In Sec. 4, we illustrate how a string abstraction can benefit from the expressiveness of the specification language in order to provide better invariants.
- We implemented the analysis in Mopsa and we modeled over 1000 library functions. In Sec. 5, we report on the results of analyzing some Juliet benchmarks and Coreutils programs. More particularly, we show how our analysis combines several symbolic domains in order to analyze C programs with an unbounded number of command-line arguments with arbitrary lengths. To our knowledge, Mopsa is the first static analyzer to perform such an analysis.

$$
\begin{array}{lll}
\mathit{stub} & ::= & (\mathit{stmt} \mid \mathit{case})* \\
\mathit{case} & ::= & \texttt{case}\ \texttt{\{}\ \mathit{stmt}*\ \texttt{\}} \\
\mathit{stmt} & ::= & \mathit{effect} \mid \mathit{cond} \\
\mathit{effect} & ::= & \texttt{alloc} : \mathit{type}\ \mathit{ident}\ = \texttt{new}\ \mathit{ident}; \\
 & \mid & \texttt{assigns} : \mathit{expr}\ [\mathit{expr}, \mathit{expr}]?; \\
 & \mid & \texttt{free} : \mathit{expr}; \\
 & & \\
\mathit{cond} & ::= & \texttt{assumes} : \mathit{form}; \\
 & \mid & \texttt{requires} : \mathit{form}; \\
 & \mid & \texttt{ensures} : \mathit{form}; \\
 & & \\
\mathit{ntype} & ::= & \texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long} \mid \texttt{float} \\
\mathit{stype} & ::= & \mathit{ntype} \mid \texttt{ptr} \\
\mathit{type} & ::= & \mathit{stype} \\
 & \mid & \mathit{type}\texttt{[}n\texttt{]}, n \in \mathbb{N} \\
 & \mid & \texttt{struct}\ \texttt{\{}\ \mathit{type}\ \mathit{ident}; \ldots \texttt{\}} \\
 & \mid & \texttt{union}\ \texttt{\{}\ \mathit{type}\ \mathit{ident}; \ldots \texttt{\}}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{form} & ::= & \mathit{expr} \diamond \mathit{expr}, \diamond \in \{\ \texttt{==}, \texttt{!=}, \ldots\ \} \\
 & \mid & \mathit{expr} \in \mathit{set} \\
 & \mid & \texttt{alive}(\mathit{expr}) \\
 & \mid & \mathit{form} \wedge \mathit{form} \\
 & \mid & \mathit{form} \vee \mathit{form} \\
 & \mid & \neg \mathit{form} \\
 & \mid & \forall\ \mathit{ident} \in [\mathit{expr}, \mathit{expr}] : \mathit{form} \\
 & \mid & \exists\ \mathit{ident} \in [\mathit{expr}, \mathit{expr}] : \mathit{form} \\
 & & \\
\mathit{set} & ::= & [\mathit{expr}, \mathit{expr}] \mid \mathit{ident} \\
\mathit{expr} & ::= & c, c \in \mathbb{R} \\
 & \mid & \texttt{\&}\mathit{ident} \\
 & \mid & \texttt{*}\mathit{expr} \\
 & \mid & \mathit{expr} \diamond \mathit{expr}, \diamond \in \{\ \texttt{+}, \texttt{-}, \ldots\ \} \\
 & \mid & \texttt{size}(\mathit{expr}) \\
 & \mid & \texttt{base}(\mathit{expr}) \\
 & \mid & \texttt{offset}(\mathit{expr})
\end{array}
$$

Fig. 2. Syntax of the modeling language.

Limitations. The following features are not supported by our analysis: recursive functions, longjumps, bitfields, inline assembly, concurrency and multi-dimensional variable length arrays.

2 Syntax and Concrete Semantics

We define the syntax and operational concrete semantics of the modeling language. The syntax is inspired from existing specification languages, such as ACSL [3] and JML [18], with the addition of resource management. The semantics expresses a relation between program states before the function call and after.

2.1 Syntax

The syntax is presented in Fig. 2. It features two kinds of statements:

– *Side-effect statements* specify the part of the input state which is modified by the function: `assigns` specifies that a variable (or an array slice) is modified by the function; `alloc` creates a fresh resource instance of a specified class (*ident*) and assigns its address to a local variable; conversely, `free` destroys a previously allocated resource. Any memory portion that is not explicitly mentioned by these statements is implicitly assumed to be unchanged. Resources model dynamic objects, such as memory blocks managed by `malloc`, `realloc` and `free`, or file descriptors managed by `open` and `close`. The models of these functions can be found in App. A. Assigning a class to resources allows supporting different attributes (e.g., read-only memory blocks) and allocation semantics (e.g., returning the lowest available integer when allocating a descriptor, which is needed to model faithfully the `dup` function).

– *Condition statements* express pre and post-conditions: `requires` defines mandatory conditions on the input environment for the function to behave correctly; `assumes` defines assumptions, and is used for case analysis; `ensures` expresses conditions on the output environment (the return value, the value of modified variables, and the size and initial state of allocated resources).

Cases. We support a disjunctive construct `case` (akin to Frama-C's *behaviors*) to describe functions with several possible behaviors. Each case is independently analyzed, after which they are all joined. Statements placed outside cases are common to all cases, which is useful to factor specification. For the sake of clarity, we will focus on the formalization of stubs without cases.

Formulas and expressions. Formulas are classic first-order, with conjunctions, disjunctions, negations and quantifiers. The atoms are C expressions (without function call nor side-effect), extended with a few built-in functions and predicates: $e \in \mathit{set}$ restricts the range of a numeric value or the class of a resource; `alive`(e) checks whether a resource has not been freed; given a pointer e, `base`(e) returns a pointer to the beginning of the memory block containing e, `size`(e) is the block size, and `offset`(e) is the byte-offset of e in the block.

2.2 Environments

Concrete memories are defined classically. The memory is decomposed into blocks: $\mathcal{B} \stackrel{\text{def}}{=} \mathcal{V} \cup \mathcal{A}$, which can be either variables in $\mathcal{V}$ or heap addresses in $\mathcal{A}$. Each block is decomposed into scalar elements in $\mathcal{S} \subseteq \mathcal{B} \times \mathbb{N} \times \mathit{stype}$, where $\wr b, o, \tau \int \in \mathcal{S}$ denotes the memory region in block b starting at offset o and having type τ. A scalar element of type τ can have values in $\mathbb{V}_\tau$, where $\mathbb{V}_\tau$ is $\mathbb{R}$ for numeric types and $\mathbb{V}_{\texttt{ptr}} \stackrel{\text{def}}{=} \mathcal{B} \times \mathbb{N}$ is a block-offset pair for pointers.[3] The set of all scalar values is $\mathbb{V} \stackrel{\text{def}}{=} \mathbb{R} \cup (\mathcal{B} \times \mathbb{N})$.

Environments, in $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{M} \times \mathcal{R}$, encode the state of the program using: a *memory environment* in $\mathcal{M} \stackrel{\text{def}}{=} \mathcal{S} \to \mathbb{V}$, mapping scalar elements to values, and a *resource environment* in $\mathcal{R} \stackrel{\text{def}}{=} \mathcal{A} \rightharpoonup (\mathit{ident} \times \mathbb{N} \times \mathbb{B})$, which is a partial map mapping allocated resources to their class, size, and liveness status (as a Boolean).

Example 1. Given the declaration: `struct s { int id; char *data; } v`, the environment:

$$\left(\begin{array}{cc} \wr v, 0, \texttt{int} \int \mapsto 5 & \wr v, 4, \texttt{ptr} \int \mapsto (@, 0) \\ \wr @, 0, \texttt{short} \int \mapsto 3 & \wr @, 2, \texttt{short} \int \mapsto -1 \end{array} , @ \mapsto (\texttt{malloc}, 4, \mathit{true}) \right)$$

encodes the state where field `v.id` has value 5 and `v.data` points to a `malloc` resource containing two `short` elements with values 3 and -1 respectively.

[3] To simplify the presentation, we assume that $\mathcal{S}$ is given (e.g. using block types) and omit `NULL` and invalid pointers. In practice, our analysis uses the dynamic cell decomposition from [19] to fully handle C pointers, union types, and type-punning.

$$
\begin{array}{lcl}
\mathbb{E}[\![\,.\,]\!] \in \mathit{expr} \to \mathcal{E} \to \mathcal{P}(\mathbb{V}) & & \\
\mathbb{E}[\![\,\texttt{size}(e)\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,\texttt{sizeof}(b) \mid (b,-) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma) \wedge b \in \mathcal{V}\,\} \\
& & \cup\,\{\,n \mid (b,-) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma) \wedge b \in \mathcal{A} \wedge (-,n,-) = \sigma(b)\,\} \\
\mathbb{E}[\![\,\texttt{base}(e)\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,b \mid (b,-) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma)\,\} \\
\mathbb{E}[\![\,\texttt{offset}(e)\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,o \mid (-,o) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma)\,\} \\
\mathbb{E}[\![\,n\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,n\,\} \\
\mathbb{E}[\![\,\&v\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,(v,0)\,\} \\
\mathbb{E}[\![\,*e\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,\rho(\lfloor b,o,\texttt{typeof}(*e)\rfloor) \mid (b,o) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma)\,\} \\
\mathbb{E}[\![\,e_1 \diamond e_2\,]\!](\rho,\sigma) & \stackrel{\text{def}}{=} & \{\,v_1 \diamond v_2 \mid v_1 \in \mathbb{E}[\![\,e_1\,]\!](\rho,\sigma) \wedge v_2 \in \mathbb{E}[\![\,e_2\,]\!](\rho,\sigma)\,\}
\end{array}
$$

Fig. 3. Concrete semantics of expressions.

$$
\begin{array}{lcl}
\mathbb{F}[\![\,.\,]\!] \in \mathit{form} \to \mathcal{P}(\mathcal{E}) & & \\
\mathbb{F}[\![\,e \in R\,]\!] & \stackrel{\text{def}}{=} & \{\,(\rho,\sigma) \mid (b,-) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma) \wedge b \in \mathcal{A} \wedge \sigma(b) = (R,-,-)\,\} \\
\mathbb{F}[\![\,e \in [a,b]\,]\!] & \stackrel{\text{def}}{=} & \{\,(\rho,\sigma) \mid \\
& & \quad n \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma) \wedge l \in \mathbb{E}[\![\,a\,]\!](\rho,\sigma) \wedge u \in \mathbb{E}[\![\,b\,]\!](\rho,\sigma) \wedge n \in [l,u]\,\} \\
\mathbb{F}[\![\,\texttt{alive}(e)\,]\!] & \stackrel{\text{def}}{=} & \{\,(\rho,\sigma) \mid (b,-) \in \mathbb{E}[\![\,e\,]\!](\rho,\sigma) \wedge b \in \mathcal{A} \wedge \sigma(b) = (-,-,\mathit{true})\,\} \\
\mathbb{F}[\![\,e_1 \diamond e_2\,]\!] & \stackrel{\text{def}}{=} & \{\,(\rho,\sigma) \mid n_1 \in \mathbb{E}[\![\,e_1\,]\!](\rho,\sigma) \wedge n_2 \in \mathbb{E}[\![\,e_2\,]\!](\rho,\sigma) \wedge n_1 \diamond n_2\,\} \\
\mathbb{F}[\![\,\neg f\,]\!] & \stackrel{\text{def}}{=} & \mathbb{F}[\![\,\mathit{de\text{-}morgan\text{-}negation}(f)\,]\!] \\
\mathbb{F}[\![\,f_1 \wedge f_2\,]\!] & \stackrel{\text{def}}{=} & \mathbb{F}[\![\,f_1\,]\!] \cap \mathbb{F}[\![\,f_2\,]\!] \\
\mathbb{F}[\![\,f_1 \vee f_2\,]\!] & \stackrel{\text{def}}{=} & \mathbb{F}[\![\,f_1\,]\!] \cup \mathbb{F}[\![\,f_2\,]\!] \\
\mathbb{F}[\![\,\forall v \in [a,b] : f\,]\!] & \stackrel{\text{def}}{=} & \{\,(\rho,\sigma) \mid \\
& & \quad l \in \mathbb{E}[\![\,a\,]\!](\rho,\sigma) \wedge u \in \mathbb{E}[\![\,b\,]\!](\rho,\sigma) \wedge (\rho,\sigma) \in \bigcap_{i\in[l,u]} \mathbb{F}[\![\,f[v/i]\,]\!]\,\} \\
\mathbb{F}[\![\,\exists v \in [a,b] : f\,]\!] & \stackrel{\text{def}}{=} & \{\,(\rho,\sigma) \mid \\
& & \quad l \in \mathbb{E}[\![\,a\,]\!](\rho,\sigma) \wedge u \in \mathbb{E}[\![\,b\,]\!](\rho,\sigma) \wedge (\rho,\sigma) \in \bigcup_{i\in[l,u]} \mathbb{F}[\![\,f[v/i]\,]\!]\,\}
\end{array}
$$

Fig. 4. Concrete semantics of formulas.

2.3 Evaluation

Expressions. The evaluation of expressions, given as $\mathbb{E}[\![\,.\,]\!] \in \mathit{expr} \to \mathcal{E} \to \mathcal{P}(\mathbb{V})$, returns the set of possible values to handle possible non-determinism (such as reading random values). It is defined by induction on the syntax, as depicted in Fig. 3. The stub builtin `size` reduces to the C builtin `sizeof` for variables and returns the size stored in the resource map for dynamically allocated blocks. Calls to `base` and `offset` evaluate their pointer argument and extract the first (respectively second) component. To simplify the presentation, we do not give the explicit definition of the C operators, which is complex but standard. Likewise, we omit a precise treatment of invalid and `NULL` pointers (see [19] for a more complete definition). Finally, we omit here reporting of C run-time errors.

Formulas. The semantics of formulas $\mathbb{F}[\![\,.\,]\!] \in \mathit{form} \to \mathcal{P}(\mathcal{E})$, shown in Fig. 4, returns the set of environments that satisfy it. It is standard, except for built-in predicates: to verify the predicate $e \in R$ (resp. `alive`(e)), we resolve the instance pointed by e and look up the resource map to check that its class equals R (resp. its liveness flag is *true*).

$$
\begin{array}{lcl}
\tilde{\mathbb{E}}[\![\,.\,]\!] \in \tilde{expr} \to \mathcal{E}\times\mathcal{E} \to \mathcal{P}(\mathbb{V}) & & \\
\tilde{\mathbb{E}}[\![\,*e\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \mathbb{E}[\![\,*e\,]\!]\,\varepsilon \\
\tilde{\mathbb{E}}[\![\,(*e)'\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \mathbb{E}[\![\,*e\,]\!]\,\varepsilon' \\
\tilde{\mathbb{E}}[\![\,\texttt{size}(e)\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \mathbb{E}[\![\,\texttt{size}(e)\,]\!]\,\varepsilon \\
\tilde{\mathbb{E}}[\![\,\texttt{size}(e')\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \mathbb{E}[\![\,\texttt{size}(e)\,]\!]\,\varepsilon' \\
\tilde{\mathbb{E}}[\![\,n\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \{\,n\,\} \\
\tilde{\mathbb{E}}[\![\,\&v\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \{\,(v,0)\,\} \\
\tilde{\mathbb{E}}[\![\,e_1\diamond e_2\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \{\,v_1\diamond v_2 \mid v_1\in\tilde{\mathbb{E}}[\![\,e_1\,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge v_2\in\tilde{\mathbb{E}}[\![\,e_2\,]\!]\,\langle\varepsilon,\varepsilon'\rangle\,\} \\
\tilde{\mathbb{E}}[\![\,\texttt{base}(e)\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \{\,b \mid (b,-)\in\tilde{\mathbb{E}}[\![\,e\,]\!]\,\langle\varepsilon,\varepsilon'\rangle\,\} \\
\tilde{\mathbb{E}}[\![\,\texttt{offset}(e)\,]\!]\,\langle\varepsilon,\varepsilon'\rangle & \stackrel{\text{def}}{=} & \{\,o \mid (-,o)\in\tilde{\mathbb{E}}[\![\,e\,]\!]\,\langle\varepsilon,\varepsilon'\rangle\,\}
\end{array}
$$

Fig. 5. Concrete semantics of relational expressions.

2.4 Relational semantics

Statements express some information on pre and post-conditions, that is, on the relation between input and output environments.

Expressions and formulas. To allow expressions to mention both the input and output state, we use the classic prime notation: e' denotes the value of expression e in the post-state. Denoting $\tilde{expr}$ the set of expressions with primes, their semantic on an input-output environment pair is given by $\tilde{\mathbb{E}}[\![\,.\,]\!] \in \tilde{expr} \to \mathcal{E}\times\mathcal{E} \to \mathcal{P}(\mathbb{V})$. Fig. 5 presents the most interesting cases: evaluating a primed dereference $\tilde{\mathbb{E}}[\![\,(*e)'\,]\!]\,\langle\varepsilon,\varepsilon'\rangle$ reduces to the non-relational evaluation $\mathbb{E}[\![\,*e\,]\!]$ on ε', while a non-primed dereference reduces to $\mathbb{E}[\![\,*e\,]\!]$ on ε. The case of $\texttt{size}(e')$ and $\texttt{size}(e)$ is similar. Other cases are analog to non-relational evaluation.

We denote by $\tilde{form}$ formulas with primes, and define their evaluation function $\tilde{\mathbb{F}}[\![\,.\,]\!] \in \tilde{form} \to \mathcal{P}(\mathcal{E}\times\mathcal{E})$ as returning a relation. As shown in Fig. 6, to evaluate predicates $e \in R$ and $\texttt{alive}(e)$, only input environments are inspected, as the resource class is an immutable property and the liveness flag can be changed only by `free` statements in previous calls. The remaining definitions are similar to the non-relational case.

Example 2. Consider again variable `v` shown in Example 1 and the following relational formula: $\texttt{v.data}' == \texttt{v.data} + 1 \wedge \texttt{*(v.data + 1)}' == 10$. When applied on the previous environment we obtain the relation:

$$
\left\langle
\begin{pmatrix}
\wr v,0,\texttt{int}\int \mapsto 5 & \wr v,4,\texttt{ptr}\int \mapsto (@,0) \\
\wr @,0,\texttt{short}\int \mapsto 3 & \wr @,2,\texttt{short}\int \mapsto -1
\end{pmatrix}, @\mapsto(\texttt{malloc},4,\mathit{true})
\right)
,
\left(
\begin{pmatrix}
\wr v,0,\texttt{int}\int \mapsto 5 & \wr v,4,\texttt{ptr}\int \mapsto (@,\mathbf{2}) \\
\wr @,0,\texttt{short}\int \mapsto 3 & \wr @,2,\texttt{short}\int \mapsto \mathbf{10}
\end{pmatrix}, @\mapsto(\texttt{malloc},4,\mathit{true})
\right)
\right\rangle
$$

Side-effect statements. We model side-effect statements as relation transformers, $\mathbb{S}_{\text{effect}}[\![\,.\,]\!] \in \mathit{effect} \to \mathcal{P}(\mathcal{E}\times\mathcal{E}) \to \mathcal{P}(\mathcal{E}\times\mathcal{E})$ shown in Fig. 7. Given an input-output relation as argument, it returns a new relation where the output part is

$$
\begin{array}{lcl}
\tilde{\mathbb{F}}[\![\,.\,]\!] \in \tilde{form} \to \mathcal{P}(\mathcal{E}\times\mathcal{E}) & & \\
\tilde{\mathbb{F}}[\![\, e \in R \,]\!] & \stackrel{\text{def}}{=} & \{\, \langle\varepsilon,\varepsilon'\rangle \mid \varepsilon \in \mathbb{F}[\![\, e \in R \,]\!] \,\} \\
\tilde{\mathbb{F}}[\![\, e \in [a,b] \,]\!] & \stackrel{\text{def}}{=} & \{\, \langle\varepsilon,\varepsilon'\rangle \mid \\
 & & \quad n \in \tilde{\mathbb{E}}[\![\, e \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge l \in \tilde{\mathbb{E}}[\![\, a \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge u \in \tilde{\mathbb{E}}[\![\, b \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge n \in [l,u] \,\} \\
\tilde{\mathbb{F}}[\![\, \texttt{alive}(e) \,]\!] & \stackrel{\text{def}}{=} & \{\, \langle\varepsilon,\varepsilon'\rangle \mid \varepsilon \in \mathbb{F}[\![\, \texttt{alive}(e) \,]\!] \,\} \\
\tilde{\mathbb{F}}[\![\, e_1 \diamond e_2 \,]\!] & \stackrel{\text{def}}{=} & \{\, \langle\varepsilon,\varepsilon'\rangle \mid v_1 \in \tilde{\mathbb{E}}[\![\, e_1 \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge v_2 \in \tilde{\mathbb{E}}[\![\, e_2 \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge v_1 \diamond v_2 \,\} \\
\tilde{\mathbb{F}}[\![\, \neg f \,]\!] & \stackrel{\text{def}}{=} & \tilde{\mathbb{F}}[\![\, \textit{de-morgan-negation}(f) \,]\!] \\
\tilde{\mathbb{F}}[\![\, f_1 \wedge f_2 \,]\!] & \stackrel{\text{def}}{=} & \tilde{\mathbb{F}}[\![\, f_1 \,]\!] \cap \tilde{\mathbb{F}}[\![\, f_2 \,]\!] \\
\tilde{\mathbb{F}}[\![\, f_1 \vee f_2 \,]\!] & \stackrel{\text{def}}{=} & \tilde{\mathbb{F}}[\![\, f_1 \,]\!] \cup \tilde{\mathbb{F}}[\![\, f_2 \,]\!] \\
\tilde{\mathbb{F}}[\![\, \forall v \in [a,b] : f \,]\!] & \stackrel{\text{def}}{=} & \{\, \langle\varepsilon,\varepsilon'\rangle \mid \\
 & & \quad l \in \tilde{\mathbb{E}}[\![\, a \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge u \in \tilde{\mathbb{E}}[\![\, b \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge \langle\varepsilon,\varepsilon'\rangle \in \bigcap_{i\in[l,u]} \tilde{\mathbb{F}}[\![\, f[v/i] \,]\!] \,\} \\
\tilde{\mathbb{F}}[\![\, \exists v \in [a,b] : f \,]\!] & \stackrel{\text{def}}{=} & \{\, \langle\varepsilon,\varepsilon'\rangle \mid \\
 & & \quad l \in \tilde{\mathbb{E}}[\![\, a \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge u \in \tilde{\mathbb{E}}[\![\, b \,]\!]\,\langle\varepsilon,\varepsilon'\rangle \wedge \langle\varepsilon,\varepsilon'\rangle \in \bigcup_{i\in[l,u]} \tilde{\mathbb{F}}[\![\, f[v/i] \,]\!] \,\}
\end{array}
$$

Fig. 6. Concrete semantics of relational formulas.

updated to take into account the effect of the statement. Thus, starting from the identity relation, by composing these statements, we can construct a relation mapping each input environment to a corresponding environment with resources allocated or freed, and variables modified. The statement $\texttt{alloc} : \tau* \; v = \texttt{new}\; R$ allocates a new instance of resource class R and assigns its address to variable v. The function $\texttt{scalars} \in \textit{type} \to \mathcal{P}(\mathbb{N} \times \textit{stype})$ returns the set of scalar types and their offsets within a given type. We have no information on the block size (except that it is a non-null multiple of the size of τ) nor the block contents; both information can be provided later using an `ensures` statement. The statement $\texttt{assigns} : e[a,b]$ modifies the memory block pointed by e and fills the elements located between indices a and b with unspecified values. Finally, $\texttt{free} : e$ frees the resource pointed by e by updating its liveness flag. These statements only use non-primed variables, hence, all expressions are evaluated in the input part of the relation, which is left intact by these transformers.

Condition statements. A condition statement adds a constraint to the initial input-output relation built by the side-effect statements. We define their semantics as a function $\mathbb{S}_{\text{cond}}[\![\,.\,]\!] \in \textit{cond} \to \mathcal{P}(\mathcal{E}\times\mathcal{E})$. Another role of these statements is to detect specification violation (unsatisfied `requires`). Thus, we enrich the set of output environments with an error state Ω, so that $\langle\varepsilon,\Omega\rangle$ denotes an input environment ε that does not satisfy a pre-condition. The semantics is given in Fig. 7. Both `assumes` and `requires` statements use the simple filter $\mathbb{F}[\![\,.\,]\!]$ as they operate on input environments. In contrast, `ensures` statements express relations between the input and the output and use therefore the relational filter $\tilde{\mathbb{F}}[\![\,.\,]\!]$. Combining two conditions is a little more subtle than intersecting their relations, due to the error state. We define a combination operator $\fatsemi$ that preserves errors detected by conditions. Due to errors, conditions are not commutative. Indeed $\texttt{assumes} : x > 0; \texttt{requires} : x \neq 0;$ is not equivalent to $\texttt{requires} : x \neq 0; \texttt{assumes} : x > 0$, as the later will report errors when $x \neq 0$.

$$\mathbb{S}_{\text{effect}}[\![\,.\,]\!] \in \mathit{effect} \to \mathcal{P}(\mathcal{E}\times\mathcal{E}) \to \mathcal{P}(\mathcal{E}\times\mathcal{E})$$
$$\mathbb{S}_{\text{effect}}[\![\,\texttt{alloc} : \tau\texttt{*}\ x = \texttt{new}\ R;\,]\!]X \stackrel{\text{def}}{=}$$
$$\{\,\langle\varepsilon, (\rho'[x\mapsto(@,0), c_{1,1}\mapsto v_{1,1},\ldots,c_{n,m}\mapsto v_{n,m}], \sigma'[@\mapsto(R, n.\texttt{sizeof}(\tau), \mathit{true})])\rangle \mid$$
$$\langle\varepsilon,(\rho',\sigma')\rangle \in X \wedge @\notin \mathit{dom}(\sigma') \wedge n\in\mathbb{N}^\star \wedge \{\,(o_1,\tau_1),\ldots,(o_m,\tau_m)\,\} = \texttt{scalars}(\tau)$$
$$\wedge\,\forall i\in[1,n], j\in[1,m] : c_{i,j} = \wr @, o_j + (i-1)\texttt{sizeof}(\tau), \tau_j \int \wedge v_{i,j}\in\mathbb{V}_{\tau_{i,j}}\,\}$$
$$\mathbb{S}_{\text{effect}}[\![\,\texttt{assigns} : e[a,b];\,]\!]X \stackrel{\text{def}}{=}$$
$$\{\,\langle\varepsilon,(\rho'[c_1\mapsto v_1,\ldots,c_{u-l+1}\mapsto v_{u-l+1}],\sigma')\rangle \mid \langle\varepsilon,(\rho',\sigma')\rangle\in X$$
$$\wedge\,(b,o)\in\mathbb{E}[\![\,e\,]\!](\rho,\sigma) \wedge l\in\mathbb{E}[\![\,a\,]\!](\rho,\sigma) \wedge u\in\mathbb{E}[\![\,b\,]\!](\rho,\sigma) \wedge \tau = \texttt{typeof}(\texttt{*}e)$$
$$\wedge\,\forall k\in[1,l-u+1] : c_k = \wr b, o+(k-1)\texttt{sizeof}(\tau), \tau \int \wedge v_k\in\mathbb{V}_\tau\,\}$$
$$\mathbb{S}_{\text{effect}}[\![\,\texttt{free} : e;\,]\!]X \stackrel{\text{def}}{=}$$
$$\{\,\langle\varepsilon,(\rho',\sigma'[@\mapsto(R,n,\mathit{false})])\rangle \mid \langle\varepsilon,(\rho',\sigma')\rangle\in X \wedge (@,-)\in\mathbb{E}[\![\,e\,]\!]\varepsilon\,\}$$
$$\mathbb{S}_{\text{effect}}[\![\,s_1; s_2;\,]\!]X \stackrel{\text{def}}{=} \mathbb{S}_{\text{effect}}[\![\,s_2\,]\!] \circ \mathbb{S}_{\text{effect}}[\![\,s_1\,]\!]\,X$$

$$\mathbb{S}_{\text{cond}}[\![\,.\,]\!] \in \mathit{cond} \to \mathcal{P}(\mathcal{E}\times\mathcal{E})$$
$$\mathbb{S}_{\text{cond}}[\![\,\texttt{assumes} : f;\,]\!] \stackrel{\text{def}}{=} \{\,\langle\varepsilon,\varepsilon'\rangle \mid \varepsilon\in\mathbb{F}[\![\,f\,]\!]\,\}$$
$$\mathbb{S}_{\text{cond}}[\![\,\texttt{requires} : f;\,]\!] \stackrel{\text{def}}{=} \{\,\langle\varepsilon,\varepsilon'\rangle \mid \varepsilon\in\mathbb{F}[\![\,f\,]\!]\,\} \cup \{\,\langle\varepsilon,\Omega\rangle \mid \varepsilon\in\mathbb{F}[\![\,\neg f\,]\!]\,\}$$
$$\mathbb{S}_{\text{cond}}[\![\,\texttt{ensures} : f;\,]\!] \stackrel{\text{def}}{=} \tilde{\mathbb{F}}[\![\,f\,]\!]$$
$$\mathbb{S}_{\text{cond}}[\![\,s_1; s_2;\,]\!] \stackrel{\text{def}}{=} \mathbb{S}_{\text{cond}}[\![\,s_1\,]\!] \fatsemi \mathbb{S}_{\text{cond}}[\![\,s_1\,]\!]$$
$$R_1 \fatsemi R_2 \stackrel{\text{def}}{=} R_1\cap R_2 \cup \{\,\langle\varepsilon,\Omega\rangle \mid \langle\varepsilon,\Omega\rangle\in R_1\,\} \cup \{\,\langle\varepsilon,\Omega\rangle \mid \langle\varepsilon,\Omega\rangle\in R_2 \wedge \langle\varepsilon,-\rangle\in R_1\,\}$$

Fig. 7. Concrete semantics of statements.

$$\begin{aligned}
&\mathbb{S}[\![\,.\,]\!] \in \mathit{stub} \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})\times\mathcal{P}(\mathcal{E})\\
&\mathbb{S}[\![\,\mathit{body}\,]\!]I \stackrel{\text{def}}{=} \text{let } R_0 = \{\,\langle\varepsilon,\varepsilon\rangle \mid \varepsilon\in I\,\} \text{ in}\\
&\qquad\qquad \text{let } R_1 = \mathbb{S}_{\text{effect}}[\![\,\texttt{effects}(\mathit{body})\,]\!]\,R_0 \text{ in}\\
&\qquad\qquad \text{let } R_2 = R_1 \fatsemi \mathbb{S}_{\text{cond}}[\![\,\texttt{conditions}(\mathit{body})\,]\!] \text{ in}\\
&\qquad\qquad \text{let } O = \{\,\varepsilon' \mid \langle-,\varepsilon'\rangle\in R_2 \wedge \varepsilon'\neq\Omega\,\} \text{ in}\\
&\qquad\qquad \text{let } X = \{\,\varepsilon \mid \langle\varepsilon,\Omega\rangle\in R_2\,\} \text{ in}\\
&\qquad\qquad (O,X)
\end{aligned}$$

Fig. 8. Concrete semantics of the stub.

Iterator. Fig. 8 shows the semantic function $\mathbb{S}[\![\,.\,]\!] \in \mathit{stub} \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})\times\mathcal{P}(\mathcal{E})$ of a complete stub. It first executes its side-effect statements only $\texttt{effects}(\mathit{body})$, then condition statements $\texttt{conditions}(\mathit{body})$, and finally applies the resulting relation R_2 to the initial states at function entry I. It returns two sets of environments: the environments O at function exit when pre-conditions are met, and the environments X at function entry that result in a violation of a pre-condition.

3 Generic Abstract Semantics

We show how an existing abstract domain for C can be extended to abstract the concrete semantics of our stubs in a generic way. The next section will focus on specific abstractions exploiting more finely the structure of stub statements.

3.1 Abstract Domain

C domain. We assume we are given an abstract domain $\mathcal{M}^\sharp$ of memories $\mathcal{P}(\mathcal{M})$ with the standard operators: least element $\bot_{\mathcal{M}}$, join $\sqcup_{\mathcal{M}}$, and widening $\triangledown_{\mathcal{M}}$, as well as a sound abstraction $\mathbb{S}^\sharp_{\mathrm{M}}[\![\,.\,]\!] \in \mathit{stmt}_{\mathcal{M}} \to \mathcal{M}^\sharp \to \mathcal{M}^\sharp$ for classic memory statement $\mathit{stmt}_{\mathcal{M}}$, including: $x \leftarrow y$, to model assignments of C expressions; $\mathit{forget}(b, x, y)$, to assign random values to a byte slice $[x, y]$ of a memory block b; $\mathit{add}(b)$, to add a memory block with random values; $\mathit{remove}(b)$ to remove a memory block; and the array sumarization operators $\mathit{expand}(b_1, b_2)$ and $\mathit{fold}(b_1, b_2)$ from [14]. $\mathit{expand}(b_1, b_2)$ creates a *weak copy* b_2 of block b_1, *i.e.* both b_1 and b_2 have the same constraints without being equal. For example, executing $\mathit{expand}(x, z)$ when $x \geq y \wedge x \in [1, 10]$ yields $x \geq y \wedge x \in [1, 10] \wedge z \geq y \wedge z \in [1, 10]$. The converse operation, $\mathit{fold}(b_1, b_2)$, creates a summary in b_1 by keeping only the constraints also implied by b_2, and then removes b_2. We exploit them to abstract unbounded memory allocation and perform weak updates.

Heap abstraction. We also assume that we are given an abstraction of heap addresses $\mathcal{P}(\mathcal{A})$ into a finite set $\mathcal{A}^\sharp$ of abstract addresses, with least element $\bot_{\mathcal{A}}$ and join $\sqcup_{\mathcal{A}}$. Classic examples include call-site abstraction, and the recency abstraction [2] we use in our implementation. An abstract address may represent a single concrete address or a (possibly unbounded) collection of addresses, which is indicated by a cardinality operator $\|.\|_{\mathcal{A}} \in \mathcal{A}^\sharp \to \{\, \mathit{single}, \mathit{many} \,\}$. Finally, we assume the domain provides an allocation function $\mathbb{A}^\sharp[\![\,.\,]\!] \in \mathcal{P}(\mathcal{A}^\sharp) \times \mathcal{M}^\sharp \to \mathcal{A}^\sharp \times \mathcal{M}^\sharp$. As an abstract allocation may cause memory blocks to be expanded or folded, and the pointers to point to different abstract addresses, the function also returns an updated memory environment.

Environments. For each abstract block in $\mathcal{A}^\sharp$, we maintain its byte size in a numeric variable $\mathsf{size}^\sharp \in \mathcal{A}^\sharp \to \mathcal{B}$ in the memory environment, and track its possible resource classes in $\mathcal{P}(\mathcal{C})$, and possible liveness status in the boolean lattice $\mathcal{P}(\{\mathit{true}, \mathit{false}\})$. The abstraction $\mathcal{E}^\sharp$ of environment sets $\mathcal{P}(\mathcal{E})$ is thus:

$$\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathcal{M}^\sharp \times \mathcal{A}^\sharp \to (\mathcal{P}(\mathcal{C}) \times \mathcal{P}(\{\mathit{true}, \mathit{false}\})) \tag{1}$$

The $\bot_{\mathcal{E}}$, $\sqcup_{\mathcal{E}}$, and $\triangledown_{\mathcal{E}}$ operators are derived naturally from those in $\mathcal{M}^\sharp$ and $\mathcal{A}^\sharp$, and we lift C statements to $\mathbb{S}^\sharp_{\mathrm{C}}[\![\, s \,]\!](\rho^\sharp, \sigma^\sharp) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp_{\mathrm{M}}[\![\, s \,]\!]\rho^\sharp, \sigma^\sharp)$.

3.2 Evaluations

Our abstraction leverages the modular architecture and the communication mechanisms introduced in the Mopsa framework [16]. We will employ notably *symbolic and disjunctive evaluations*, which we recall briefly.

Expressions. In the concrete semantics, expressions are evaluated into values. Abstracting expression evaluation as functions returning abstract values, such as intervals, would limit the analysis to non-relational properties. Instead, domains

in Mopsa can evaluate expressions into other expressions: based on the current abstract state, expression parts are simplified into more abstract ones that other domains can process. A common example is relying on abstract variables. For instance, the memory domain will replace a `size`(e) expression into the variable $\mathsf{size}^\sharp(b)$ after determining that e points to block b, producing a purely numeric expression. Communicating expressions ensures a low coupling between domains, while preserving relational information (e.g., `size`$(e) < i$ reduces to comparing two numeric variables, $\mathsf{size}^\sharp(b)$ and i). A domain can also perform a case analysis and transform one expression into a disjunction of several expressions, associated to a partition of the abstract state (e.g., if e can point to several blocks). Formally, a domain $\mathcal{D}^\sharp$ implements expression evaluation as a function: $\phi \in expr \to \mathcal{D}^\sharp \to \mathcal{P}(expr \times \mathcal{D}^\sharp)$. To express concisely that the rest of the abstract computation should be performed in parallel on each expression and then joined, we define here (and use in our implementation) a monadic bind operator:

$$\begin{aligned} &\mathrm{let}^\sharp_\sqcup\ (f, Y^\sharp) \in \phi[e]X^\sharp \text{ in } body \stackrel{\text{def}}{=} \\ &\quad \bigsqcup_{(g,Z^\sharp)\in\phi[e]X^\sharp} body[f/g, Y^\sharp/Z^\sharp] \end{aligned} \tag{2}$$

We illustrate formally abstract expression evaluation $\mathbb{E}^\sharp[\![\,.\,]\!]$ on the `size`(e) expression. First, the pointer domain handles the pointer expression e: $\mathbb{E}^\sharp[\![\,e\,]\!]\varepsilon^\sharp$ returns a set of triples (b, o, ε') where b is an abstract block, o a numeric offset expression, and ε' the part of ε where e points into block b. Thanks to this disjunction, the abstract semantics of `size`(e) follows closely the concrete one:

$$\begin{aligned} \mathbb{E}^\sharp[\![\,\texttt{size}(e)\,]\!]\varepsilon^\sharp \stackrel{\text{def}}{=}\ & \mathrm{let}^\sharp_\sqcup\ ((b, -), \varepsilon_1^\sharp) \in \mathbb{E}^\sharp[\![\,e\,]\!]\,\varepsilon^\sharp \text{ in} \\ & \text{if } b \in \mathcal{V} \text{ then } \{\,(\texttt{sizeof}(b), \varepsilon_1^\sharp)\,\} \\ & \text{else } \{\,(\mathsf{size}^\sharp(b), \varepsilon_1^\sharp)\,\} \end{aligned} \tag{3}$$

Formulas. Evaluation of formulas is defined by the function $\mathbb{F}^\sharp[\![\,.\,]\!] \in formula \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp$, shown in Fig. 9. We focus on the most interesting cases which are the quantified formulas. Existential quantification reduces to assigning to v the interval $[a, b]$ and keeping only environments that satisfy f. Universal quantification are handled very similarly to a loop `for(v=a; v<=b; v++) assume(f)`. We perform an iteration with widening for v from a to b and we over-approximate the sequence of states statisfying f. The overall formula is satisfied for states reaching the end of the sequence. These generic transfer functions can be imprecise in practice. We will show later that specific domains can implement natively more precise transfer functions for selected quantified formulas.

Relations. The concrete semantics requires evaluating expressions and formulas not only on states, by also on relations. To represent relations in the abstract, we simply introduce a family of primed variables: $\mathsf{primed}^\sharp \in \mathcal{B} \to \mathcal{B}$ returns the primed version of a block (i.e., the block in the post-state). This classic technique allows lifting any state domain to a relation domain. Combined with relational domains, we can express complex relationships between values in the pre- and the post-state, if needed. The relation abstractions $\tilde{\mathbb{E}}^\sharp[\![\,.\,]\!]$ and $\tilde{\mathbb{F}}^\sharp[\![\,.\,]\!]$ of $\tilde{\mathbb{E}}[\![\,.\,]\!]$ and

$$
\begin{array}{l}
\mathbb{F}^\sharp[\![\,.\,]\!] \in \mathit{form} \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp \\
\mathbb{F}^\sharp[\![\,\texttt{alive}(e)\,]\!](\rho^\sharp,\sigma^\sharp) \stackrel{\text{def}}{=} \\
\quad \mathrm{let}^\sharp_{\sqcup_\mathcal{E}}\ ((b,-),(\rho^\sharp,\sigma^\sharp)) \in\ \mathbb{E}^\sharp[\![\,e\,]\!]\,(\rho^\sharp,\sigma^\sharp)\ \mathrm{in} \\
\quad \mathrm{if}\ b \notin \mathcal{A}^\sharp\ \mathrm{then}\ \bot\ \mathrm{else} \\
\quad \mathrm{let}\ (C,f) = \sigma^\sharp(b)\ \mathrm{in} \\
\quad \mathrm{if}\ f = \{\mathit{false}\}\ \mathrm{then}\ \bot\ \mathrm{else} \\
\quad \mathrm{let}\ f' = \mathrm{if}\ \|b\|_\mathcal{A} = \mathit{single}\ \mathrm{then}\ \{\mathit{true}\}\ \mathrm{else}\ f\ \mathrm{in} \\
\quad (\rho^\sharp,\sigma^\sharp[b \mapsto (C,f')]) \\
\mathbb{F}^\sharp[\![\,e \in R\,]\!](\rho^\sharp,\sigma^\sharp) \stackrel{\text{def}}{=} \\
\quad \mathrm{let}^\sharp_{\sqcup_\mathcal{E}}\ ((b,-),(\rho^\sharp,\sigma^\sharp)) \in\ \mathbb{E}^\sharp[\![\,e\,]\!]\,(\rho^\sharp,\sigma^\sharp)\ \mathrm{in} \\
\quad \mathrm{if}\ b \notin \mathcal{A}^\sharp\ \mathrm{then}\ \bot\ \mathrm{else} \\
\quad \mathrm{let}\ (C,f) = \sigma^\sharp(b)\ \mathrm{in} \\
\quad \mathrm{if}\ R \notin C\ \mathrm{then}\ \bot\ \mathrm{else} \\
\quad \mathrm{let}\ C' = \mathrm{if}\ \|b\|_\mathcal{A}\| = \mathit{single}\ \mathrm{then}\ \{R\}\ \mathrm{else}\ C\ \mathrm{in} \\
\quad (\rho^\sharp,\sigma^\sharp[b \mapsto (C',f)]) \\
\mathbb{F}^\sharp[\![\,\exists v \in [a,b] : f\,]\!]\varepsilon^\sharp \stackrel{\text{def}}{=}\ \mathbb{S}^\sharp_\mathrm{C}[\![\,\mathit{remove}(v)\,]\!] \circ \mathbb{F}^\sharp[\![\,f\,]\!] \circ \mathbb{S}^\sharp_\mathrm{C}[\![\,v \leftarrow [a,b]\,]\!] \circ \mathbb{S}^\sharp_\mathrm{C}[\![\,\mathit{add}(v)\,]\!]\,\varepsilon^\sharp \\
\mathbb{F}^\sharp[\![\,\forall v \in [a,b] : f\,]\!]\varepsilon^\sharp \stackrel{\text{def}}{=} \\
\quad \mathrm{let}\ \varepsilon^\sharp_0 = \mathbb{F}^\sharp[\![\,v \le b\,]\!] \circ \mathbb{S}^\sharp_\mathrm{C}[\![\,v \leftarrow a\,]\!] \circ \mathbb{S}^\sharp_\mathrm{C}[\![\,\mathit{add}(v)\,]\!]\,\varepsilon^\sharp\ \mathrm{in} \\
\quad \mathrm{let}\ \varepsilon^\sharp_1 = \mathrm{lfp}\,\lambda X.\, X \triangledown_\mathcal{E} (\varepsilon^\sharp_0 \sqcup_\mathcal{E} \mathbb{S}^\sharp_\mathrm{C}[\![\,v \leftarrow v+1\,]\!] \circ \mathbb{F}^\sharp[\![\,f\,]\!] \circ \mathbb{F}^\sharp[\![\,v \le b\,]\!]X)\ \mathrm{in} \\
\quad \mathbb{S}^\sharp_\mathrm{C}[\![\,\mathit{remove}(v)\,]\!] \circ \mathbb{F}^\sharp[\![\,v > b\,]\!]\,\varepsilon^\sharp_1
\end{array}
$$

Fig. 9. Abstract semantics of formulas.

$\tilde{\mathbb{F}}[\![\,.\,]\!]$ can be easily expressed in terms of the state abstractions $\mathbb{E}^\sharp[\![\,.\,]\!]$ and $\mathbb{F}^\sharp[\![\,.\,]\!]$ we already defined. As an example, the evaluation of a primed dereference $(*e)'$ simply evaluates e into a set of memory blocks b and offset expressions o, and outputs a dereference of the primed block $\mathsf{primed}^\sharp(b)$ at the (non-primed) offset expression o, which can be handled by the (relation-unaware) memory domain:

$$
\begin{array}{l}
\tilde{\mathbb{E}}^\sharp[\![\,(*e)'\,]\!]\varepsilon^\sharp \stackrel{\text{def}}{=}\ \mathrm{let}^\sharp_\sqcup\ ((b,o),\varepsilon^\sharp_1) \in\ \tilde{\mathbb{E}}^\sharp[\![\,e\,]\!]\,\varepsilon^\sharp\ \mathrm{in} \\
\qquad\qquad \{\,(*(\texttt{typeof}(e)))((\texttt{char*})\&\mathsf{primed}^\sharp(b)+o),\varepsilon^\sharp_1)\,\}
\end{array}
\tag{4}
$$

3.3 Transfer Functions

Side-effect statements. The effect of a statement is approximated by $\mathbb{S}^\sharp_\mathrm{effect}[\![\,.\,]\!] \in \mathit{effect} \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ defined in Fig. 10. Resource allocation $\texttt{alloc} : v = \texttt{new}\ R$ first asks the underlying heap abstraction for a new abstract address with $\mathbb{A}^\sharp[\![\,.\,]\!]$, which is bound to a new variable v; a new size variable $\mathsf{size}^\sharp$ is created and the resource map is updated with the class and liveness information. The block is also initialized with random values using *forget*. Assignments $\texttt{assigns} : e[x,y]$ reduces to *forget* on the primed version of the block b e points to (recall that the output value is specified by a later `ensures`). Finally, $\texttt{free} : e$ resets the liveness flag of the primed block.

Condition statements. The abstract semantics of condition statements is given by $\mathbb{S}^\sharp_\mathrm{cond}[\![\,.\,]\!] \in \mathit{cond} \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp \times \mathcal{E}^\sharp$, defined in Fig. 10. The function returns a pair of abstract environments: the first one over-approximates the output environments satisfying the condition, while the second one over-approximates

$$\mathbb{S}^\sharp_{\text{effect}}[\![\,.\,]\!] \in \mathit{effect} \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp$$
$$\mathbb{S}^\sharp_{\text{effect}}[\![\,\texttt{alloc} : v = \texttt{new}\ R\,]\!](\rho^\sharp, \sigma^\sharp) \overset{\text{def}}{=}$$
$$\quad \text{let } (@, \rho_1^\sharp) = \mathbb{A}^\sharp[\![\, \mathit{dom}(\sigma^\sharp)\,]\!]\ \rho^\sharp \text{ in}$$
$$\quad \text{let } \sigma_2^\sharp = \sigma_1^\sharp[@ \mapsto (\{\, R\,\}, \{\, \mathit{true}\,\})] \text{ in}$$
$$\quad \text{let } \varepsilon_2^\sharp = \mathbb{S}^\sharp_{\text{C}}[\![\, v \leftarrow @\,]\!] \circ \mathbb{S}^\sharp_{\text{C}}[\![\, \mathit{add}(v)\,]\!]\ (\rho_1^\sharp, \sigma_2^\sharp) \text{ in}$$
$$\quad \text{let } \varepsilon_3^\sharp = \mathbb{S}^\sharp_{\text{C}}[\![\, \mathsf{size}^\sharp(@) \geq 0\,]\!] \circ \mathbb{S}^\sharp_{\text{C}}[\![\, \mathit{add}(\mathsf{size}^\sharp(@))\,]\!]\ \varepsilon_2^\sharp \text{ in}$$
$$\quad \mathbb{S}^\sharp_{\text{C}}[\![\, \mathit{forget}(@, 0, \mathsf{size}^\sharp(@) - 1)\,]\!]\ \varepsilon_3^\sharp$$
$$\mathbb{S}^\sharp_{\text{effect}}[\![\,\texttt{assigns} : e[x, y]\,]\!]\varepsilon^\sharp \overset{\text{def}}{=}$$
$$\quad \text{let}^\sharp_{\sqcup_\mathcal{E}}\ (b, o), \varepsilon_1^\sharp \in \mathbb{E}^\sharp[\![\, e\,]\!]\ \varepsilon^\sharp \text{ in}$$
$$\quad \text{let } n = \texttt{sizeof}(*e) \text{ in}$$
$$\quad \mathbb{S}^\sharp_{\text{C}}[\![\, \mathit{forget}(\mathsf{primed}^\sharp(b), o + x \times n), o + y \times n\,]\!]\ \varepsilon_1^\sharp$$
$$\mathbb{S}^\sharp_{\text{effect}}[\![\,\texttt{free} : e\,]\!]\varepsilon^\sharp \overset{\text{def}}{=}$$
$$\quad \text{let}^\sharp_{\sqcup_\mathcal{E}}\ (b, -), (\rho_1^\sharp, \sigma_1^\sharp) \in \mathbb{E}^\sharp[\![\, e\,]\!]\ \varepsilon^\sharp \text{ in}$$
$$\quad \text{if } b \notin \mathcal{A}^\sharp \text{ then } \bot_\mathcal{E} \text{ else}$$
$$\quad \text{let } C, f = \sigma_1^\sharp(b) \text{ in}$$
$$\quad \text{if } \|b\|_\mathcal{A} = \mathit{single} \text{ then } (\rho_1^\sharp, \sigma_1^\sharp[\mathsf{primed}^\sharp(b) \mapsto (C, \{\, \mathit{false}\,\})])$$
$$\quad \text{else } (\rho_1^\sharp, \sigma_1^\sharp[\mathsf{primed}^\sharp(b) \mapsto (C, f \cup \{\, \mathit{false}\,\})])$$

$$\mathbb{S}^\sharp_{\text{cond}}[\![\,.\,]\!] \in \mathit{cond} \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp \times \mathcal{E}^\sharp$$
$$\mathbb{S}^\sharp_{\text{cond}}[\![\,\texttt{assumes} : f;\,]\!]\varepsilon^\sharp \overset{\text{def}}{=}$$
$$\quad (\mathbb{F}^\sharp[\![\, f\,]\!]\varepsilon^\sharp, \bot_\mathcal{E})$$
$$\mathbb{S}^\sharp_{\text{cond}}[\![\,\texttt{requires} : f;\,]\!]\varepsilon^\sharp \overset{\text{def}}{=}$$
$$\quad (\mathbb{F}^\sharp[\![\, f\,]\!]\varepsilon^\sharp, \mathbb{F}^\sharp[\![\, \neg f\,]\!]\varepsilon^\sharp)$$
$$\mathbb{S}^\sharp_{\text{cond}}[\![\,\texttt{ensures} : f;\,]\!]\varepsilon^\sharp \overset{\text{def}}{=}$$
$$\quad (\mathbb{F}^\sharp[\![\, f\,]\!]\varepsilon^\sharp, \bot_\mathcal{E})$$
$$\mathbb{S}^\sharp_{\text{cond}}[\![\, s_1; s_2;\,]\!]\varepsilon^\sharp \overset{\text{def}}{=}$$
$$\quad \text{let } (\varepsilon_1^\sharp, \omega_1^\sharp) = \mathbb{S}^\sharp_{\text{cond}}[\![\, s_1\,]\!]\ \varepsilon^\sharp \text{ in}$$
$$\quad \text{let } (\varepsilon_2^\sharp, \omega_2^\sharp) = \mathbb{S}^\sharp_{\text{cond}}[\![\, s_2\,]\!]\ \varepsilon_1^\sharp \text{ in}$$
$$\quad (\varepsilon_2^\sharp, \omega_1^\sharp \sqcup_\mathcal{E} \omega_2^\sharp)$$

Fig. 10. Abstract semantics of statements.

the input environments violating mandatory conditions specified with `requires` statements.

Iterator. The abstract semantic of a whole stub is defined in Fig. 11. First, the *expand* function is used to construct an identity relation over the input abstract state $\varepsilon_0^\sharp$. To improve efficiency, this is limited to the blocks that are effectively modified by the stub; this set is over-approximated using the `assigned` function, which resolves the pointer expressions occurring in `assigns` statement. Then, side-effect statements are evaluated. Note that, for an `assigns` : $a[x, y]$ statement, while whole blocks pointed by a are duplicated in the output state, only the parts in the $[x, y]$ range are assigned random values. Condition statements are then executed, collecting contract violation and refining the output state. Finally, we remove the unprimed version of primed (i.e., modified) blocks and the primed block into its unprimed version, thus reverting to a state abstraction that models the output state. In case of a primed block b modeling several concrete blocks (i.e., $\|b\|_\mathcal{A} = \mathit{many}$), the primed block is folded into the unprimed version, so as to preserve the values before the call, resulting in a weak update.

4 Specific Abstract Semantics: the Case of C Strings

We now show how we can design a formula-aware abstract domain, with an application to C string analysis. The domain handles precisely selective quantified formula, while reverting in the other cases (as all other domains) to the generic operators.

$$
\begin{array}{l}
\mathbb{S}^\sharp[\![\,.\,]\!] \in \mathit{stub} \to \mathcal{E}^\sharp \to \mathcal{E}^\sharp \times \mathcal{E}^\sharp \\
\mathbb{S}^\sharp[\![\,\mathit{body}\,]\!]\varepsilon_0^\sharp \stackrel{\text{def}}{=} \\
\quad \text{let } (a_1, \ldots, a_n) = \texttt{assigned}(\mathit{body}) \text{ in} \\
\quad \text{let}^\sharp_{\sqcup_\mathcal{E}} ((b_1, -), \varepsilon_1^\sharp) \in \mathbb{E}^\sharp[\![\,a_1\,]\!]\,\varepsilon_0^\sharp \text{ in} \\
\quad \ldots \\
\quad \text{let}^\sharp_{\sqcup_\mathcal{E}} ((b_n, -), \varepsilon_n^\sharp) \in \mathbb{E}^\sharp[\![\,a_n\,]\!]\,\varepsilon_{n-1}^\sharp \text{ in} \\
\quad \text{let } \tilde{\varepsilon}_0^\sharp = \mathbb{S}^\sharp_{\mathrm{C}}[\![\,\mathit{prime}(b_n)\,]\!] \circ \cdots \circ \mathbb{S}^\sharp_{\mathrm{C}}[\![\,\mathit{prime}(b_1)\,]\!]\varepsilon_n^\sharp \text{ in} \\
\quad \text{let } \tilde{\varepsilon}_1^\sharp = \mathbb{S}^\sharp_{\mathrm{effect}}[\![\,\texttt{effects}(\mathit{body})\,]\!]\,\tilde{\varepsilon}_0^\sharp \text{ in} \\
\quad \text{let } \tilde{\varepsilon}_2^\sharp, \omega^\sharp = \mathbb{S}^\sharp_{\mathrm{cond}}[\![\,\texttt{conditions}(\mathit{body})\,]\!]\,\tilde{\varepsilon}_1^\sharp \text{ in} \\
\quad \text{let } \varepsilon^\sharp = \mathbb{S}^\sharp_{\mathrm{C}}[\![\,\mathit{unprime}(b_n)\,]\!] \circ \cdots \circ \mathbb{S}^\sharp_{\mathrm{C}}[\![\,\mathit{unprime}(b_1)\,]\!]\tilde{\varepsilon}_2^\sharp \text{ in} \\
\quad (\varepsilon^\sharp, \omega^\sharp) \\
\text{where:} \\
\quad \mathit{prime}(b) \stackrel{\text{def}}{=} \mathit{expand}(b, \mathsf{primed}^\sharp(b)) \\
\quad \mathit{unprime}(b) \stackrel{\text{def}}{=} \\
\quad\quad \text{if } b \in \mathcal{A}^\sharp \wedge \|b\|_\mathcal{A} = \mathit{many} \text{ then } \mathit{fold}(\mathsf{primed}^\sharp(b), b) \text{ else } \mathit{rename}(\mathsf{primed}^\sharp(b), b)
\end{array}
$$

Fig. 11. Abstract semantics of the stub.

String length domain. Strings in C are arrays of `char` elements containing a delimiting null byte `'\0'` indicating the end of the string. Many functions in the standard C library take strings as arguments and assume they contain a null byte delimiter. We want to express and check this assumption in the function stubs. We exploit a classic abstraction already present in Mopsa: the StringLength domain [17] that maintains a numeric abstract variable $\mathsf{length}^\sharp \in \mathcal{B} \to \mathcal{B}$ for arrays to store the offset of the first null byte. It thus infers, for each array a, an invariant of the form:

$$\forall i \in [0, \mathsf{length}^\sharp(a) - 1] : a[i] \neq 0 \wedge a[\mathsf{length}^\sharp(a)] = 0 \tag{5}$$

Example 3. Consider the following example, where `n` ranges in $[0, 99]$:

```
1  for (int i = 0; i < n; i++) a[i] = 'x';
2  a[n] = '\0';
```

An analysis with the Intervals domain will infer that $\mathsf{length}^\sharp(a) \in [0, 99]$. Adding the Polyhedra domain, we will moreover infer that $\mathsf{length}^\sharp(a) = n$.

Stub transfer functions. Within a stub, a pre-condition stating the validity of a string pointed to by an argument named s is naturally expressed as:

$$\texttt{requires} : \exists i \in [0, \texttt{size}(s) - \texttt{offset}(s) - 1] : s[i] \texttt{ == } 0; \tag{6}$$

Proving this requirement requires checking the emptiness of its negation, which involves a universal quantifier. Using the generic abstraction from last section, it is equivalent to proving emptiness after the loop `for (i = 0; i < size(s)-offset(s); i++) s[i] != 0`. This, in turn, requires an iteration with widening and, unless s has constant length, a relational domain with sufficient precision, which is costly.

Formula	Case	Condition	State transformer
$\exists i \in [lo, hi] : s[i]$ == 0	#1	$\mathsf{length}^\sharp(s) > hi$	$\lambda\epsilon^\sharp.\ \bot$
	#2	$\mathsf{length}^\sharp(s) \leq hi$	$\lambda\epsilon^\sharp.\ \epsilon^\sharp$
$\forall i \in [lo, hi] : s[i]$!= 0	#1	$\mathsf{length}^\sharp(s) \notin [lo, hi]$	$\lambda\epsilon^\sharp.\ \epsilon^\sharp$
	#2	$\mathsf{length}^\sharp(s) \in [lo, hi]$	$\lambda\epsilon^\sharp.\ \bot$

Table 1. Transfer functions of formulas in the string length domain.

To solve these problems, we propose a direct interpretation of both formulas in the string domain, i.e., we add transfer functions for $\mathbb{F}^\sharp[\![\ \exists i \in [lo, hi] : s[i]$ == 0 $]\!]$ and $\mathbb{F}^\sharp[\![\ \forall i \in [lo, hi] : s[i]$!= 0 $]\!]$,[4] as shown in Table. 1. They perform a case analysis: the abstract state $\epsilon^\sharp$ is split into two cases according to a condition, and we keep all environments in one case ($\lambda\epsilon^\sharp.\ \epsilon$) and none in the other ($\lambda\epsilon^\sharp.\ \bot$). For instance, assuming that (5) holds, then Case #1 of $\mathbb{F}^\sharp[\![\ \exists i \in [lo, hi] : s[i]$ == 0 $]\!]$ states that the quantification range $[lo, hi]$ covers only elements before the null byte, so that the formula does not hold. Case #2 states that there is a value in $[lo, hi]$ greater than or equal to the string length, in which case $s[i]$ may be null and the formula may be valid. Similarly, Case #1 of $\mathbb{F}^\sharp[\![\ \forall i \in [lo, hi] : s[i]$!= 0 $]\!]$ arises when the null byte is outside the quantification range, so that the formula may be valid. In Case #2, the null byte is in the range, and the formula is definitely invalid. We stress on the fact that all the conditions are interpreted symbolically in the numeric domain; hence, lo and hi are not limited to constants, but can be arbitrary expressions.

Example 4. Let us illustrate how the predicate (6) can be verified on the following abstract environment:

$$\epsilon^\sharp = \begin{pmatrix} \wr s, 0, \texttt{ptr} \int \mapsto \{\,(@, 0)\,\} \\ \mathsf{size}^\sharp(@) \geq 1 \\ \mathsf{length}^\sharp(@) \in [0, \mathsf{size}^\sharp(@) - 1] \end{pmatrix}, @ \mapsto (\{\,\texttt{malloc}\,\}, \mathit{true}) \tag{7}$$

which represents the case of a variable s pointing to a resource instance @ allocated by `malloc` with at least one byte. The string domain indicates that the position of the null byte is between 0 and $\mathsf{size}^\sharp(@) - 1$. When checking the formula $\exists i \in [0, \texttt{size}(s) - \texttt{offset}(s) - 1] : s[i]$ == 0, the condition for Case #1 never holds since:

$$(\texttt{size}(s) - \texttt{offset}(s) - 1 = \mathsf{size}^\sharp(@) - 1) \wedge (\mathsf{length}^\sharp(@) \leq \mathsf{size}^\sharp(@) - 1)$$

When checking its negation, $\forall i \in [0, \texttt{size}(s) - \texttt{offset}(s) - 1] : s[i]$!= 0, Case #1 is also unsatisfiable, for the same reason. As the transformer for Case #2 returns $\bot$, the overall result is $\bot$, proving that Requirement (6) holds: the stub does not raise any alarm.

[4] We actually support the comparison of $s[i]$ with arbitrary expressions. We limit the description to the case of comparisons with 0 for the sake of clarity.

Genericity of formulas. An important motivation for using a logic language is to exploit its expressiveness within abstract domains to analyze several stubs with the same transfer functions. We show that this is indeed the case: the transfer function that was used to validate strings in the previous section can be used, without modification, to compute string lengths.

Example 5. Let us go back to the example of the `strlen` function defined as:

```
1  /*$
2   * requires: s != NULL ∧ offset(s) ∈ [0, size(s));
3   * requires: ∃i ∈ [0, size(s)-offset(s)): s[i] == 0;
4   * ensures : return ∈ [0, size(s)-offset(s));
5   * ensures : s[return] == 0;
6   * ensures : ∀i ∈ [0, return): s[i] != 0;
7   */
8  size_t strlen(const char s);
```

and consider again the environment (7). As shown before, the `requires` statements at line 3 validating the string do not raise any alarm. At line 5, the classic transfer functions of the StringLength domain [17] infer that:

$$0 \leq \mathsf{length}^\sharp(@) \leq \texttt{return}$$

since $s[\texttt{return}] = 0$ and $\mathsf{length}^\sharp(@)$ is the position of the first null byte. Finally, at line 6, both cases of the second transfer function in Table 1 are valid. Since we keep a non-$\bot$ post-state only for Case #1, we obtain:

$$\begin{aligned} & 0 \leq \mathsf{length}^\sharp(@) \leq \texttt{return} \wedge \mathsf{length}^\sharp(@) \notin [0, \texttt{return} - 1] \\ \Leftrightarrow\ & 0 \leq \mathsf{length}^\sharp(@) \leq \texttt{return} \wedge \mathsf{length}^\sharp(@) > \texttt{return} - 1 \\ \Leftrightarrow\ & 0 \leq \mathsf{length}^\sharp(@) = \texttt{return} \end{aligned}$$

hence the domain precisely infers that `strlen` returns the length of string @.

5 Experiments

We implemented our analysis in the Mopsa framework [16]. It consists of 29503 lines of OCaml code (excluding parsers). Among them, 16449 lines (56%) are common with analyses of other languages, such as Python. C domains consist of 11342 lines (38%) and the stub abstraction consists of 1712 lines (6%).

We wrote 14191 lines of stub, modeling 1108 functions from 50 headers from the Glibc implementation of the standard C library, version 8.28 [13]. All stubs thoroughly check their arguments (pointers, strings, integers, floats), soundly model their side effects, dynamic memory allocation, open files and descriptors. We refrained form implicit assumptions, such as non-aliasing arguments. At an average of 8 meaningful lines per stub, the language proved to be concise enough. Some examples can be found in App. A.

To assess the efficiency and the precision of our implementation, we target two families of programs. We run our analysis on part of NIST Juliet Tests Suite [5], a large collection of small programs with artificially injected errors. These tests

Code	Title	Tests	Lines	Time (h:m:s)	✅	⚠
CWE121	Stack-based Buffer Overflow	2508	234k	00:59:12	26%	74%
CWE122	Heap-based Buffer Overflow	1556	174k	00:37:12	28%	72%
CWE124	Buffer Underwrite	758	93k	00:18:28	86%	14%
CWE126	Buffer Over-read	600	75k	00:14:45	40%	60%
CWE127	Buffer Under-read	758	89k	00:18:26	87%	13%
CWE190	Integer Overflow	3420	440k	01:24:47	52%	48%
CWE191	Integer Underflow	2622	340k	01:02:27	55%	45%
CWE369	Divide By Zero	497	109k	00:13:17	55%	45%
CWE415	Double Free	190	17k	00:04:21	100%	0%
CWE416	Use After Free	118	14k	00:02:40	99%	1%
CWE469	Illegal Pointer Subtraction	18	1k	00:00:24	100%	0%
CWE476	NULL Pointer Dereference	216	21k	00:04:53	100%	0%

Table 2. Analysis results on Juliet. ✅: precise analysis, ⚠: analysis with false alarms.

are precious to reveal soundness bugs in analyzers, but do not reflect real-world code bases. Hence, we also target more realistic programs from the Coreutils package [12], which are widely used command-line utilities. These programs, while not very large, depend heavily on the C standard library. We run all our tests on an Intel Xeon 3.40GHz processor running Linux.

5.1 Juliet

The Juliet Tests Suite [5] is organized using the Common Weakness Enumeration taxonomy [1]. It consists of a large number of tests for each CWE. Each test contains *bad* and *good* functions. Bad functions contain one instance of the CWE, while good functions are safe.

We selected 12 categories from NIST Juliet 1.3 matching the safety violations detected by Mopsa. For each test, we have analyzed the good and the bad functions and measured the analysis time and the number of reported alarms. Three outcomes are possible. The analysis is *precise* if it reports (*i*) exactly one alarm in the bad function that corresponds to the tested CWE, and (*ii*) no alarm in the good function. The analysis is *unsound* if no alarm is reported in the bad function. Otherwise, the analysis is *imprecise*.

The obtained results are summarized in Table 2. From each category, we have excluded tests that contain unsupported features or that do not correspond to runtime errors. As expected, all analyses are sound: Mopsa detects the target CWE in every bad test. However, half of the tests were imprecise. Much of this imprecision comes from the gap between Mopsa's error reporting and the CWE taxonomy. For instance, an invalid string passed to a library function may be reported as a stub violation while Juliet expects a buffer overflow. By considering precise an analysis reporting no alarm in the good function and exactly one alarm in the bad function (without considering its nature), the overall precision increases to 71% (e.g. 89% of CWE121 tests become precise). Other factors also contribute to the imprecisions, such as the lack of disjunctive domains. Finally, many tests use the socket API to introduce non-determinism, and the current

file management abstraction was not precise enough to prove the validity of some file descriptors.

5.2 Coreutils

Our second benchmark includes 19 out of 106 programs from Coreutils version 8.30 [12]. Each program consists in a principal C file containing the `main` function, and library functions that are shared among all Coreutils programs. Due to these complex dependencies, it was difficult to extract the number of lines corresponding to individual programs. Instead, we computed the number of atomic statements, consisting of assignments and tests (e.g. in `for`, `while` and `switch` statements), in the functions reached by the analysis. This gives an indication of the size of the program, but the scale is not comparable with line count metrics.

Scenarios. Three scenarios were considered. The first one consists in analyzing the function `main` without any argument. In the second scenario, we call `main` with one symbolic argument with arbitrary size. The last scenario is the most general: `main` is called with a symbolic number of symbolic arguments.

Abstractions. For each scenario, four abstractions were compared. In the first abstraction A_1, we employed the Cells memory domain [19] over the Intervals domain. The second abstraction A_2 enriches A_1 with the StringLength domain [17] improved as discussed in Sect. 4. The third abstraction A_3 enriches A_2 with the Polyhedra domain [8,15] with a static packing strategy [4]. Finally, A_4 enriches A_3 with a PointerSentinel domain that tracks the position of the first `NULL` pointer in an array of pointers; it is similar to the string length domain and useful to represent a symbolic `argv` and handle functions such as `getopt` (see App. A.4).

Limitations. The analysis of recursive calls is not yet implemented in Mopsa. We have found only one recursive function in the analyzed programs, which we replaced with a stub model. The second limitation concerns the `getopt` family of functions. We have not considered the case where these functions modify the `argv` array by rearranging its elements in some specific order, since such modifications make the analysis too imprecise. However, we believe that this kind of operation can be handled by an enhanced PointerSentinel domain. This is left as future work.

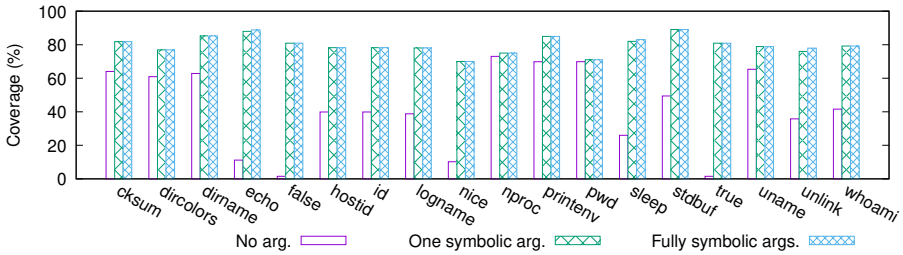
Precision. Table 12a shows the number of alarms for every analysis. The most advanced abstraction A_4 reduces significantly the number of alarms, specially for the fully symbolic scenario. This gain is not due to one specific abstraction, but it comes from the cooperation of several domains, most notably between Polyhedra and StringLength. This also emphasizes the effectiveness of domain communication mechanisms within Mopsa [16], notably symbolic expression evaluation.

Program	Statements	No arg.				One symbolic arg.				Fully symbolic args.			
		A_1	A_2	A_3	A_4	A_1	A_2	A_3	A_4	A_1	A_2	A_3	A_4
`cksum`	292	53	29	28	36	135	106	106	53	136	107	106	53
`dircolors`	507	104	54	47	47	185	158	154	100	186	159	154	99
`dirname`	183	59	14	13	13	120	90	90	22	120	90	90	21
`echo`	241	16	3	3	3	216	179	175	33	216	179	175	34
`false`	131	0	0	0	0	89	61	61	13	89	61	61	13
`hostid`	193	25	9	8	8	91	63	63	16	92	64	63	16
`id`	193	25	9	8	8	91	63	63	16	92	64	63	16
`logname`	196	25	8	7	7	93	62	62	15	94	63	62	15
`nice`	323	16	3	3	3	145	105	104	18	151	111	105	20
`nproc`	356	81	36	35	35	136	99	99	33	137	100	99	32
`printenv`	179	70	29	28	28	159	131	130	59	161	133	130	59
`pwd`	342	81	23	20	20	116	70	68	23	116	70	68	22
`sleep`	289	25	8	7	7	125	97	97	29	128	99	97	29
`stdbuf`	546	97	53	52	52	327	269	267	125	329	271	268	127
`true`	131	0	0	0	0	89	61	61	13	89	61	61	13
`uname`	251	67	25	24	24	105	72	72	27	106	73	73	33
`unexpand`	478	149	93	92	92	226	179	179	95	226	179	179	94
`unlink`	204	25	8	7	7	98	68	68	15	103	71	68	15
`whoami`	202	27	9	8	8	95	63	63	16	96	64	63	16

(a) Number of reported alarms.

Program	No arg.				One symbolic arg.				Fully symbolic args.			
	A_1	A_2	A_3	A_4	A_1	A_2	A_3	A_4	A_1	A_2	A_3	A_4
`cksum`	12.62	15.76	46.86	46.32	33.69	39.67	175.92	174.45	34.21	39.3	174.5	193.64
`dircolors`	70.27	88.49	292.38	228.75	174.46	192.94	514.1	646.22	160.91	198.07	533.13	595.14
`dirname`	22.56	29.04	97.96	85.65	22.95	30.38	90.99	140.88	24.97	28.89	96.04	119.86
`echo`	8.73	9.12	13.38	12.48	10.74	13.52	26.03	25.44	11.44	13.24	24.75	156.15
`false`	8.72	9.17	13.38	13.45	9.33	11.35	19.63	18.9	10.05	11.26	19.54	19.18
`hostid`	9.87	10.18	21.7	20.63	14.74	16.72	41.13	53.68	14.17	16.61	42.08	53.41
`id`	9.51	11.53	22.68	20.65	13.66	16.5	43.39	55.37	13.75	18.96	40.51	54.57
`logname`	9.31	10.75	20.13	19.42	15.97	16.51	39.37	45.06	13.47	17.05	40.69	48.72
`nice`	9.26	9.08	13.64	12.57	25.42	30.04	113.35	177.38	23.98	30.73	148.1	238.55
`nproc`	23.1	30.35	103.64	90.52	25.72	32.96	110.4	150.21	25.7	34.17	112.39	128.86
`printenv`	21.43	27.63	93.83	94.08	22.82	28.34	111.41	206.16	22.52	28.06	131.27	200.63
`pwd`	23.81	29.34	95.41	84.18	24.1	29.05	88.72	127.68	22.41	29.59	98.15	113.56
`sleep`	11.48	13.11	26.93	24.77	17.54	19.86	59.62	65.49	16.64	21.42	62.27	71.32
`stdbuf`	37.23	56.73	214.48	190.39	42.37	63.34	229.52	291.24	42.32	65.75	215.85	255.32
`true`	8.73	9.13	12.57	12.08	10.89	11.27	18.64	19.4	10.04	11.62	18.95	21.63
`uname`	21.85	28.46	86.38	81.68	24.19	28.9	85.85	102.31	23.95	30.97	95.13	129.77
`unexpand`	68.75	137.73	400.55	366.1	65.14	138.18	361.77	525.35	61.9	149.1	378.31	364.11
`unlink`	11.35	12.88	26.24	27.23	14.74	16.05	40.34	49.04	16.82	18.63	49.03	58.85
`whoami`	10.51	11.17	21.28	22.17	14.98	16.13	41.89	59.91	14.27	16.69	48.57	61.3

(b) Analysis time in seconds.



(c) Coverage of abstraction A_4.

Fig. 12. Analysis results on Coreutils programs.

Efficiency. As shown in Table 12b, the gain in precision comes at the cost of degraded performances. The most significant decrease corresponds to the introduction of the Polyhedra domain. Note that our current packing strategy is naive (assigning for each function one pack holding all its local variables); a more advanced strategy could improve scalability.

Coverage. We have also measured the ratio of statements reached by the analysis in the three scenarios. While not a formal guarantee of correctness, a high level of coverage provides some reassurance that large parts of the programs are not ignored due to soundness errors in our implementation or our stubs. We discuss only the case of abstraction A_4, as other cases provide similar results. Figure 12c presents the results. In most cases, using one symbolic argument helps covering a significantly larger part of the program compared to analyzing `main` without any argument. Coverage with one or several symbolic arguments is roughly the same, possibly due to the control flow over-approximations caused by even a single symbolic argument. Nevertheless, only the last scenario, covering an unbounded number of arguments, provides a soundness guarantee that all the executions of the program are covered. As far as we know, this is not supported in the static value analyses by Frama-C [10] nor Astrée [4].

6 Conclusion

We presented a static analysis by abstract interpretation of C library functions modeled with a specification language. We defined an operational concrete semantics of the language and proposed a generic abstraction that can be supported by any abstract domain. We also showed how a C string domain could be enriched with specialized transfer functions for specific formulas appearing in stubs, greatly improving the analysis precision. We integrated the proposed solution into the Mopsa static analyzer and experimented it on Juliet benchmarks and Coreutils programs. In the future, we plan to extend our coverage of the standard C library, provide models for other well-known libraries, such as OpenSSL, and experiment on larger program analyses. In addition, we envisage to upgrade our specification language to support more expressive logic. Finally, we want to improve the quality of our results by adding more precise abstractions, such as trace partitioning, or more efficient modular iterators.

References

1. Common weakness enumeration: A community-developed list of software weakness types. `https://cwe.mitre.org/`, accessed: 2020-05-24
2. 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. Baudin, P., Cuoq, P., Fillâtre, J., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL:ANSI/ISO C Specification Language, `http://frama-c.com/acsl.html`

4. 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)
5. Black, P.E.: Juliet 1.3 test suite: Changes from 1.2. Tech. Rep. NIST TN – 1995, NIST (Jun 2018)
6. 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)
7. 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)
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. D. Bühler, P.C., Yakobowski, B.: Eva: The evolved value analysis plug-in
11. Fahndrich, M.: Static verification for code contracts. In: Proc. of SAS'10. LNCS, vol. 6337, pp. 2–5 (2010)
12. GNU: Coreutils: GNU core utilities, `https://www.gnu.org/software/coreutils/`
13. GNU: The GNU C library, `https://www.gnu.org/software/libc/`
14. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Proc. of TACAS'04. LNCS, vol. 2988, pp. 512–529. Springer (Mar 2004)
15. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 661–667. CAV '09, Springer-Verlag (2009)
16. Journault, M., Miné, A., Monat, M., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. of VSTTE'19. pp. 1–17 (2019)
17. Journault, M., Ouadjaout, A., Miné, A.: Modular static analysis of string manipulations in C programs. In: Proc. of SAS'18. LNCS (2018)
18. Leavens, G., Ruby, C., Leino, K.R.M., Poll, E., Jacobs, B.: JML: Notations and tools supporting detailed design in Java. Proc. of OOPSLA'18 pp. 105–106 (2000)
19. 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)

A Stub Examples

This appendix presents additional representative examples of the stubs we developed for the GNU C library.

A.1 Predicates

To simplify stub coding, following other logic-base specification languages, Mopsa allows defining logic predicates, that can be then used in stubs. For instance, we define the following useful predicates on C strings: `valid_string(s)` states that

s is zero-terminated, and is useful as argument precondition; `in_string(x,s)` states that x points within string s before its null character, which is useful to state post-conditions.

```
/*$
 * predicate valid_string(s):
 *   s != NULL ∧ offset(s) ∈ [0, size(s) - 1]
 *   ∧ ∃ k ∈ [0, size(s) - offset(s) - 1]: s[k] == 0;
 */

/*$
 * predicate in_string(x,s):
 *   ∃ k ∈ [0, size(s) - offset(s) - 1]:
 *     ( x == s + k
 *       ∧ ∀ l ∈ [0, k - 1]: s[l] != 0 );
 */
```

A.2 Memory Management

Memory allocation functions show examples of resource allocation, and the use of cases to simplify the specification of functions with several behaviors.

```
/*$
 * case {
 *   alloc:   void* r = new malloc;
 *   ensures: size(r) == __size;
 *   ensures: return == r;
 * }
 *
 * case {
 *   assigns: _errno;
 *   ensures: return == NULL;
 * }
 *
 * case {
 *   assumes: __size == 0;
 *   ensures: return == NULL;
 * }
 */
void *malloc (size_t __size);
```

```
/*$
 * case {
 *   assumes:  __ptr == NULL;
 * }
 *
 * case {
 *   assumes:  __ptr != NULL;
 *   requires: __ptr ∈ malloc;
```

```
 *   requires: alive(__ptr);
 *   requires: offset(__ptr) == 0;
 *   free:     __ptr;
 * }
 */
void free (void *__ptr);
```

```
/*$
 * case {
 *   assumes:  __ptr == NULL;
 *   assumes:  __size == 0;
 *   ensures: return == NULL;
 * }
 *
 * case {
 *   assumes:  __ptr == NULL;
 *   alloc:    void* r = new malloc;
 *   ensures:  size(r) == __size;
 *   ensures:  return == r;
 * }
 *
 * case {
 *   assumes:  __ptr != NULL;
 *   assumes:  __size == 0;
 *   requires: __ptr ∈ malloc;
 *   free:     __ptr;
 *   ensures: return == NULL;
 * }
 *
 * case {
 *   assumes:  __ptr != NULL;
 *   requires: __ptr ∈ malloc;
 *   local:    void* r = new malloc;
 *   ensures:  size(r) == __size;
 *   ensures:  size(__ptr) >= __size ⇒
 *             ∀ i ∈ [0, __size):
 *               ((unsigned char*)r)[i] == ((unsigned char*)__ptr)[i];
 *   ensures:  size(__ptr) <= __size ⇒
 *             ∀ i ∈ [0, size(__ptr)):
 *               ((unsigned char*)r)[i] == ((unsigned char*)__ptr)[i];
 *   free:     __ptr;
 *   ensures:  return == r;
 * }
 *
 * case {
 *   assigns: _errno;
 *   ensures: return == NULL;
 * }
 */
void *realloc (void *__ptr, size_t __size);
```

A.3 File Descriptors

File descriptors are another example of resource allocation, but use a specific class that the analyzer can track to allocate integer file descriptors according to the C library policy: the least unused integer is picked. This allows modeling precisely patterns such as `close(0); int f = open("...");`. `read` reads non-deterministic values, after checking that the file has been opened and not closed.

```
/*$
 * requires: valid_string(__file);
 *
 * case {
 *   alloc:   int fd = new FileDescriptor;
 *   ensures: return == fd;
 * }
 *
 * case {
 *   assigns: _errno;
 *   ensures: return == -1;
 * }
 */
int open (const char *__file, int __oflag, ...);
```

```
/*$
 * requires: __fd ∈ FileDescriptor;
 * requires: alive(__fp as FileDescriptor);
 * requires: size(__buf) >= offset(__buf) + __nbytes;
 *
 * case {
 *   assigns: ((char*)__buf)[0, __nbytes);
 *   ensures: return ∈ [0, __nbytes];
 * }
 *
 * case {
 *   assigns: _errno;
 *   ensures: return == -1;
 * }
 */
ssize_t read (int __fd, void *__buf, size_t __nbytes);
```

```
/*$
 * requires: __fd ∈ FileDescriptor;
 * requires: alive(__fp as FileDescriptor);
 *
 * case {
 *   free:    __fd as FileDescriptor;
 *   ensures: return == 0;
 * }
 *
 * case {
```

```
 *   assigns: _errno;
 *   ensures: return == -1;
 * }
 */
int close (int __fd);
```

A.4 Command-line Arguments

We provide the simplified model of the `getopt` function we used in Coreutil analyses.

```
/*$
 * requires: ___argc > 0;
 * requires: optind ∈ [0, ___argc];
 * requires: valid_string(__shortopts);
 * requires: ∀ i ∈ [0, ___argc - 1]: valid_string(___argv[i]);
 * assigns: optind;
 * assigns: opterr;
 * assigns: optopt;
 * assigns: optarg;
 * ensures: optind′∈[1, ___argc];
 * ensures: optarg′ != NULL ⇒ ∃ i ∈ [0, ___argc - 1]:
 *            in_string(optarg′, ___argv[i]);
 * ensures: return ∈ [-1, 255];
 * case {
 *   assigns: ___argv[0, ___argc - 1];
 *   ensures: ∀ i ∈ [0, ___argc - 1]: ∃ j ∈ [0, ___argc - 1]:
 *      (___argv[i])′ == ___argv[j];
 * }
 */
int getopt (int ___argc, char *const *___argv, const char *__shortopts);
```