Structuring Abstract Interpreters through State and Value Abstractions

Sandrine Blazy[1], David Bühler[2], and Boris Yakobowski[2]

[1] IRISA - University of Rennes 1, `sandrine.blazy@irisa.fr` ,
[2] CEA, LIST, Software Safety Lab, `{david.buhler,boris.yakobowski}@cea.fr`

Abstract. We present a new modular way to structure abstract interpreters. Modular means that new analysis domains may be plugged-in. These abstract domains can communicate through different means to achieve maximal precision. First, all abstractions work cooperatively to emit alarms that exclude the undesirable behaviors of the program. Second, the state abstract domains may exchange information through abstractions of the possible value for expressions. Those value abstractions are themselves extensible, should two domains require a novel form of cooperation. We used this approach to design EVA, an abstract interpreter for C implemented within the FRAMA-C framework. We present the domains that are available so far within EVA, and show that this communication mechanism is able to handle them seamlessly.

1 Introduction

Static analysis of C programs by abstract interpretation [9] has known considerable progress in recent years, in terms of both research breakthrough and industrial-strength implementations. Verifying C programs precisely remains of paramount importance for at least two reasons. On the one hand, C remains the choice language for safety-critical programs. Its low-level nature makes the compilation process simple enough (with non-optimizing compilers) that the equivalence between the source code and the binary produced can be checked. This is often a requirement of the qualification process. On the other hand, many programs routinely used in computers or embedded devices, and thus open to cyber-attacks, remain written in C (the Linux kernel, bind, openssl, etc).

Designing sound abstract analyzers that remain precise on large classes of programs is challenging. Having a sound analyzer generally means that a large number of false alarms are emitted. To improve precision, the analyzer can be extended with dedicated analysis *domains*, that will be better suited to handle particular code fragments. However, integrating multiple domains remains a challenge in itself. First, those domains must remain relatively independent: adding one domain should not require modifying the existing ones. However, they must also be able to cooperate, and exchange information.

In abstract interpretation, such communication is usually handled through the use of a *reduced product* [10]. However, such products are hard to define between rich domains. Moreover, reduced products are not modular, and adding a domain requires extensive modifications to existing ones. Thus, abstract analyzers often implement approximations of the reduced product [7]. Furthermore, domains often do not inter-reduce directly, but instead use a communication interface (see e.g. Astrée [11] and Verasco [16]).

The Frama-C framework [17] features an abstract interpreter called Value Analysis (abbreviated as value), that has been successfully used to verify safety-critical code [12]. Its main features are an intricate memory abstraction (able to represent efficiently and precisely both low-level concepts such as unions and bitfields, and high-level ones such as arrays), and an instance of a trace partitioning domain [18] (able to unroll loops or to analyze separately the branches of a disjunction). The abstract domain of value is not relational. Aggressive trace partitioning can be used to work around this limitation: the relational information is instead carried out by the disjunction encoded by the multiple states. Nevertheless, relational domains are desirable. Also problematic was the fact that the analyzer has been written around its domain, resulting in a very tight coupling. So far, adding new domains – relational or not – was not possible.

In this work, we go beyond what was done in value and present an abstract interpreter for C, called eva (for *E*volved *V*alue *A*nalysis). The main novelty of this analyzer —and our contribution— lies in the generic communication language between the abstract domains. This language is based on abstractions of C values, while domains are abstractions of memory states. Both state and value abstractions are extensible, and different domains may communicate through different values. Domains also cooperate to state the *alarms* about the undesirable behaviors that may occur during a program execution. Finally, abstract domains do not need to share the same abstraction for the memory, which facilitates the integration of domains with different granularities in their vision of the program.

The main contributions of our paper are the following:

– A new design for abstract interpreters and the collaboration between domains, relying on a separation between value and state abstractions.
– A semantics and a cooperative emission mechanism for the alarms that report undesirable behaviors.
– An open-source abstract interpreter for C, relying on a modular architecture aimed at easing the introduction of new abstractions.
– An implementation of multiple abstract domains, exercising the various communication mechanisms.

The rest of the paper is organized as follows. The semantics of our language is given in Section 2. We propose our new modular architecture in Section 3. Value and state abstractions are described in Sections 4 and 5. The analysis domains we have implemented are presented in Section 6. Related and future works are discussed in Sections 7 and 8.

2 Formalization of our Language

Abstract interpretation links a very precise, but generally undecidable, concrete semantics to an abstract one – the abstract semantics being a sound approx-

$$
\begin{array}{rcll}
\mathit{arith} & ::= & \texttt{char} \mid (\texttt{signed} \mid \texttt{unsigned})\ \mathit{integer} \mid \mathit{float} & \\
\mathit{scalar} & ::= & \mathit{arith} \mid \mathit{type}\ \texttt{pointer} & \\
\tau \in \mathit{type} & ::= & \mathit{scalar} \mid \tau\,[n] \ \mid \ \{\mathit{field}_i : \tau_i\}_{i \leq n} & \\
e \in \mathit{expr} & ::= & \mathit{cst} & \mathit{cst} \in \overline{\mathbb{V}} = \mathbb{Q} \ \cup \ \{(x,i) \mid x \in \mathcal{X},\ i \in \mathbb{N}\} \\
& \mid & \Diamond\,(e^n) & \Diamond \in \{+, \leq, (\tau)\,, \dots\} \\
& \mid & *_\tau\, a & \\
a \in \mathit{addr} & ::= & e \mid a.\mathit{field} \mid a[e] & \\
\mathit{stmt} & ::= & *_\tau a := e \mid e{=}{=}0? &
\end{array}
$$

Fig. 1. Language syntax

imation of the concrete one. This section defines the syntax of our language, its concrete semantics, and the properties expected from an abstract semantics. The language itself is mostly orthogonal to the rest of the paper. However, it is required to state the soundness properties of abstract transformers.

2.1 Language

Fig. 1 introduces the syntax of our language, inspired by that of Miné [20]. Programs operate over a fixed, finite set of variables $x \in \mathcal{X}$, whose types can be `char`, `signed` or `unsigned` integers, floating-point, pointers, arrays of known size or structures. Expressions e are either a scalar constant *cst*, the application of a n-ary operator $\Diamond$ to n expressions, or the dereference of an address a. A constant is either a rational (for arithmetic values) or the pair (x, i) of a variable x together with a bytes-expressed offset i (for pointers). For readability, we often write $\&x$ for $(x, 0)$. Operators on expressions include arithmetic operations, comparisons and casts between scalar types. Addresses are either a direct expression (interpreted as a pointer), or addresses plus offsets for fields in aggregates and cells in arrays. The dereference $*_\tau a$ of the address a is called a *lvalue*. The direct dereference of a variable $*_\tau(\&x)$ can be written x, as in the C syntax.

Statements are either assignments or tests that halt execution when the condition does not hold. A program P is represented by its control-flow graph, where nodes are integer-numbered program points and edges are labeled by statements. For clarity, we write our examples using a C-like syntax.

2.2 Concrete Semantics

The concrete state of a program is described by an untyped memory $\mathfrak{m} \in \mathfrak{M}$, mapping *valid* (defined later) byte locations to single-byte characters. The C standard guarantees that a character value fits in one byte. The concrete values $\overline{\mathbb{V}}_\tau$ of other scalar type τ may be encoded on successive bytes, whose number and meaning depend on the hardware architecture (which we assume known). We assume given the size `sizeof` (τ) of each scalar type, as well as a set of bijective functions ϕ_τ, each one interpreting a sequence of n bytes as a value of the scalar type τ of size n (hence of type $\left(\overline{\mathbb{V}}_{\texttt{char}}\right)^{\texttt{sizeof}(\tau)} \to \overline{\mathbb{V}}_\tau$) and conversely for their inverse. The `sizeof` function is extended to expressions.

$$\overline{\llparenthesis \Diamond \rrparenthesis} : \overline{\mathbb{V}}^n \to \overline{\mathbb{V}} + \Omega \qquad \left| \qquad \begin{aligned} \mathtt{loc}_\tau(v) &\triangleq \begin{cases} (b, i+n)_{0 \le n < \mathtt{sizeof}(\tau)} & \text{if } v = (b,i) \in \mathcal{L}_\tau \\ \Omega & \text{otherwise} \end{cases} \\ \mathfrak{m}_\tau[v] &\triangleq \phi_\tau\left(\mathfrak{m}\left(\mathtt{loc}_\tau(v)\right)\right) \end{aligned} \right.$$

$$\overline{[\![\Diamond(e^n)]\!]_{\mathfrak{m}}} \triangleq \overline{\llparenthesis \Diamond \rrparenthesis}\left(\overline{[\![e]\!]_{\mathfrak{m}}}^n\right) \qquad \left| \qquad \overline{[\![*_\tau e]\!]_{\mathfrak{m}}} \triangleq \mathfrak{m}_\tau[\overline{[\![e]\!]_{\mathfrak{m}}}] \right.$$

$$\overline{\{\!|\, *_\tau\, a := e \,|\!\}}(\mathfrak{m}) \triangleq \mathfrak{m}\left[\mathtt{loc}_\tau\left(\overline{[\![a]\!]_{\mathfrak{m}}}\right) \mapsto \phi_\tau^{-1}(\overline{[\![e]\!]_{\mathfrak{m}}})\right]$$

$$\overline{\{\!|e{=}{=}0?|\!\}}(\mathfrak{m}) \triangleq \begin{cases} \mathfrak{m} & \text{if } \overline{[\![e{=}{=}0]\!]_{\mathfrak{m}}} = 1 \\ \bot & \text{otherwise} \end{cases}$$

Fig. 2. Selected rules of the concrete semantics

Concrete values in $\overline{\mathbb{V}}$ are either arithmetic values in $\mathbb{Q}$, or pointer values. A pointer value is either the `NULL` pointer (interpreted as 0) or a pair of a variable and an offset. A *location* is a pointer value together with a type. A location of type τ is *valid* when its offset plus the size being read (i.e. the size of τ) is smaller than the size of the type of the variable. The set of valid locations of type τ is written $\mathcal{L}_\tau$. Hence, a memory $\mathfrak{m} \in \mathfrak{M}$ has actually type $\mathcal{L}_{\mathtt{char}} \to \overline{\mathbb{V}}_{\mathtt{char}}$.

Fig. 2 details some parts of the evaluation $\overline{[\![e]\!]_{\mathfrak{m}}}$ of an expression e of scalar type in memory $\mathfrak{m}$. It produces either a value in $\overline{\mathbb{V}}$ or an error Ω, in case of an illegal operation. The evaluation of $\Diamond(e^n)$ relies on a semantics $\overline{\llparenthesis \Diamond \rrparenthesis}$ from the values of the arguments to the result, that does not involve $\mathfrak{m}$. It is either defined in the C standard, or implementation-defined. The rules for the evaluation of addresses are not shown. Computing the address of an array cell $e\,[e']$ shifts the address of e by the evaluation of e', using pointer arithmetic; computing the address of a field is similar. If a pointer expression e evaluates to a valid τ-location l, its dereference $*_\tau e$ interprets the $\mathtt{sizeof}\,(\tau)$ bytes of the memory starting at location n (denoted by $\mathtt{loc}_\tau(v)$) as having type τ. Otherwise, the dereference leads to the error value.

The semantics $\overline{\{\!|\mathtt{stmt}|\!\}}$ of a statement is a transfer function over states, described in the last equations of Fig. 2. An assignment stores in the memory bytes of the lvalue the characters corresponding to the value of the right expression. A test blocks the execution, only allowing states in which the condition holds. The transfer of a statement fails if the evaluation of an expression leads to the error value. An assignment also fails if the written location is not valid.

Our concrete semantics maps each program node n to the set $\mathbb{S}(n)$ of all possible memories at this point. The semantics of the entire program P is then the smallest solution to the following equations:

$$\mathbb{S}(0) \triangleq \mathfrak{M} \qquad \mathbb{S}(j) \triangleq \bigcup_{(i,\mathtt{stmt},j)\in P} \overline{\{\!|\mathtt{stmt}|\!\}}(\mathbb{S}(i))$$

2.3 Abstract Semantics

The soundness of an abstract semantics usually relies on a concretization function γ, that connects each abstraction to the sets of concrete elements it models.

$$\begin{array}{lr} \gamma : \mathbb{X}^{\#} \to 2^{\overline{\mathbb{X}}} & x_1 \sqsubseteq x_2 \Rightarrow \gamma\,(x_1) \subseteq \gamma\,(x_2) \\ \gamma\,(\top) = 2^{\overline{\mathbb{X}}} & \gamma\,(x_1) \cup \gamma\,(x_2) \subseteq \gamma\,(x_1 \sqcup x_2) \\ \gamma\,(\bot) = \emptyset & \gamma\,(x_1) \cap \gamma\,(x_2) \subseteq \gamma\,(x_1 \sqcap x_2) \end{array}$$

Fig. 3. Soundness requirements for lattices

Then, an abstract semantics $\{\!|\cdot|\!\}^{\#}$ is a sound approximation of a concrete semantics $\overline{\{\!|\cdot|\!\}}$ if for all abstract values v, $\overline{\{\!|\gamma\,(v)|\!\}} \subseteq \gamma(\{\!|v|\!\}^{\#})$ (the semantics is implicitly lifted on sets).

It is also convenient for the abstractions to have a lattice structure. Fig. 3 presents the soundness guarantees required for a lattice $\mathbb{X}^{\#}$, with respect to the concretization. The partial order is consistent with the inclusion of concrete sets. The join $\sqcup$ and the meet $\sqcap$ over-approximate respectively the union and the intersection of sets of concrete values. $\top$ is the greatest value, whose concretization contains all concrete values. The smallest element $\bot$ denotes the abstraction with an empty concretization.

A sound lattice abstraction of concrete memory states defines an over approximation $\{\!|\texttt{stmt}|\!\}^{\#}$ of its semantics. The following equations define the abstract semantics of a program P: the soundness properties ensure that any solution is a correct approximation of its concrete semantics.

$$\mathbb{S}^{\#}\,(0) \triangleq \top \qquad\qquad \mathbb{S}^{\#}\,(j) \triangleq \bigsqcup_{(i,\texttt{stmt},j)\in P} \{\!|\texttt{stmt}|\!\}^{\#}\left(\mathbb{S}^{\#}\,(i)\right)$$

3 Architecture of a Modular Abstract Interpreter

In this section, we propose a new architecture to structure an abstract interpreter, introducing the distinction between value and state abstractions. We then describe how this architecture has been implemented in EVA to enable some interactions between abstract domains.

3.1 Hierarchy of Abstractions

We separate the abstractions on which an abstract interpreter relies into both *state* and *value* abstractions. A state abstraction represents the set of concrete states that may occur at a program point during a concrete execution. A value abstraction represents the C values an expression may have in some concrete states. The state abstractions handle the semantics of statements, while the value abstractions operate at the level of expressions. The value abstractions are the communication interface used by the state abstractions to interact with each other. Fig. 4 sketches the architecture of a modular analyzer following these principles. The services each layer provides are given on the left, and the syntax fragments on which they operate on the right.

We assume given a fixpoint engine that performs a forward analysis over a control-flow graph. It propagates the state abstractions of an *abstract domain*, inferring properties at each statement. An abstract domain has a join-semilattice

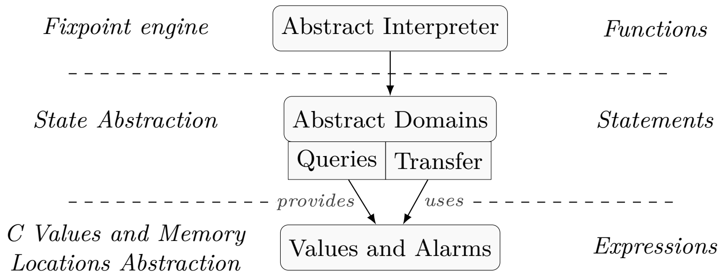

Fig. 4. Overall layers of our architecture

structure, fulfilling the properties given in Fig. 3. The join is used when two branches of the graph merge. A widening operator is mandatory, to ensure the convergence of the fixpoint computation. A domain must also provide:

- sound transformers, defining the abstract semantics of the domain. They model the effect of a statement on a state, and must satisfy the properties defined in Section 2.3.
- queries, which extract information from abstract states by assigning a *value* to some expressions. They are detailed in Section 5.

The communication between abstract domains is achieved through non-relational abstractions of *values* and *locations*. They over-approximate respectively the sets of possible C values for an expression, and the sets of possible memory locations for an address. Values and locations have a meet-semilattice structure, to intersect the values produced by multiple abstract states. They also provide sound approximations of the arithmetic operators on expressions and addresses. As the concrete operators may cause undesirable behavior at execution time, their abstract counterparts also produce *alarms*, which signal the error cases. The alarms are abstractions of the undesirable behaviors (mostly, undefined behaviors [14, Annex J.2]) that the analyzer tracks. Importantly, the alarms are part of the communication interface between abstract domains, along with values and locations. They are all formally defined in Section 4, while the interactions between domains are detailed in Section 5. In this paper, we do not distinguish values and locations further, as they fulfill the same role and the same requirements.

3.2 Communication through Value Abstractions

This design has been successfully implemented in EVA, the new modular abstract interpreter of Frama-C. Eva features a cooperative evaluation of expressions in a product of abstract states. It computes alarms and value abstractions for each expression or address involved in a statement, using the information provided by each domain. Then, all the computed abstractions are made available for the state transformers, to precisely model the effect of the statement. As these abstractions have been computed cooperatively, information may flow from a domain to another, without direct exchanges.

Within EVA, the evaluator for expressions interleaves forward and backward evaluation steps. Informally, a *forward evaluation* is a bottom-up propagation of value abstractions, from the lvalues and constants, to the root of an expression. It queries the state abstractions to extract a value for variables, and relies otherwise on the value semantics of the C operators. Conversely, a *backward evaluation* is a top-down propagation aiming at reducing the values computed for the subterms of an expression. It relies on the backward counterparts for value operations, which learn information from a result and try to reduce the arguments. Note that all domains may benefit from the reductions achieved by the evaluator.

Importantly, both states and values are extensible, and may be a combination of multiple abstractions. A generic combiner is provided for both of them. For the domain, relational and non-relational abstractions can be composed together. They interact through the shared computation of value abstractions. This may appear to prohibit the communication between abstract domains understanding different value abstractions. However, value abstractions may themselves be combined into a regular reduced product. The inter-reduction between value components indirectly achieves an inter-reduction of abstract domains working on different values, as shown by the example below.

Example 1. We consider two memory domains I and C storing information about the possible values of integer variables. I and C respectively use intervals and congruences as value abstractions. Assume a condition $x > 3$, where I and C provide respectively the interval abstraction $[0..12]$ and the congruence $0[3]$ for x. The two values for x are reduced to $[4..12]$ and $0[3]$ when backward-evaluating the condition. Then, the inter-reduction between values reduces the interval information to $[6..12]$. Finally, I can learn this more precise information for x when it abstracts the effects of the whole condition.

4 Value Abstractions

This section presents the semantics of alarms and value abstractions, that are cooperatively used to approximate the evaluation of expressions.

4.1 Alarms

An abstract interpreter emits an alarm at each program point where it fails to prove the absence of undesirable behaviors. Each alarm may reveal a real bug, or be due to the over-approximations made by some abstractions. As such, alarms are over-approximations of the undesirable behaviors of a program. They stem from illegal operations on expressions. They are produced by the abstract operators on value abstractions, accumulated during expression evaluation, and ultimately raised by the analyzer. By pointing out all the potential run-time errors, alarms are the main result of the analyzer for the end user. In order to produce as few alarms as possible, it is essential that the domains may directly influence the generation of alarms during an evaluation. Thus, in our architecture, alarms are part of the interface between the domains and the analyzer.

$$\begin{tikzcd}[column sep=small] & \texttt{unknown} \arrow[dl, dash] \arrow[dr, dash] & \\ \texttt{true} \arrow[dr, dash] & & \texttt{false} \arrow[dl, dash] \\ & \texttt{inconsistency} & \end{tikzcd}$$

$$\texttt{kind} = \texttt{closed} \mid \texttt{open}$$
$$\mathbb{A} = (\mathit{assertion} \to \mathit{status}) \times \texttt{kind}$$

$$\boldsymbol{e} \in \mathit{expr}^n \quad \boldsymbol{V} \subseteq \overline{\mathbb{V}}^n \quad \mathbf{A} \in \mathbb{A} \qquad \overline{[\![\diamondsuit\,(e^n)]\!]} : \overline{\mathbb{V}}^n \to \overline{\mathbb{V}} + \Omega$$

$$\mathbf{A} = (A, \mathit{kind}) \models_{\mathbb{A}} \overline{[\![\diamondsuit\,(\boldsymbol{e})]\!]}(\boldsymbol{V}) \Leftrightarrow$$
$$\begin{cases} \forall a \in A, \quad \forall \boldsymbol{v} \in \overline{\mathbb{V}}^n, \quad \neg a\,[\boldsymbol{e} \leftarrow \boldsymbol{v}] \;\Leftrightarrow\; \diamondsuit\,(\boldsymbol{v}) = \Omega & (1) \\ \forall a \in A, \quad A(a) = \texttt{true} \;\Rightarrow\; \forall \boldsymbol{v} \in \boldsymbol{V}, a\,[\boldsymbol{e} \leftarrow \boldsymbol{v}] & (2) \\ \forall a \in A, \quad A(a) = \texttt{false} \;\Rightarrow\; \forall \boldsymbol{v} \in \boldsymbol{V}, \neg a\,[\boldsymbol{e} \leftarrow \boldsymbol{v}] & (3) \\ \mathit{kind} = \texttt{closed} \Rightarrow \forall \boldsymbol{v} \in \boldsymbol{V},\ (\forall a \in A,\ a\,[\boldsymbol{e} \leftarrow \boldsymbol{v}]) \Rightarrow \diamondsuit\,(\boldsymbol{v}) \neq \Omega & (4) \end{cases}$$

Fig. 5. Semantics of alarms

Formally, we define alarms as maps from assertions to logical statuses ranging over `true`, `false` or `unknown`. The assertions are guards against the undesirable behaviors. If the status of an assertion is `true`, then its corresponding undesirable behavior never occurs. Otherwise, the undesirable behavior may occur (`unknown` status) or definitely happens if the program point is reachable (`false` status). The alarm maps may be `closed` or `open`: a `closed` map contains all alarms which may occur for the given expression, while an `open` map just gives a status to some of these alarms. `closed` maps are always sound abstractions of all undesirable behaviors of an expression. `open` maps are simpler to assert. They can exclude or guarantee *some* errors but offer no assurance of completeness.

Fig. 5 defines the soundness of alarm maps in $\mathbb{A}$. In the concrete semantics, C operators on expressions may return either a value in $\overline{\mathbb{V}}$ or an error Ω, in case of undesirable behaviors. Given a predicate a, $a\,[\boldsymbol{e} \leftarrow \boldsymbol{v}]$ represents the truth-value of a where expressions $\boldsymbol{e}$ are replaced by concrete values $\boldsymbol{v}$. We denote by $\mathbf{A} \models_{\mathbb{A}} \overline{[\![\diamondsuit\,(\boldsymbol{e})]\!]}(\boldsymbol{V})$ the fact that the alarms $\mathbf{A}$ are a sound abstraction of the possible undesirable behaviors of the n-ary operator $\diamondsuit$ applied to a vector $\boldsymbol{e}$ of expressions whose values are in the concrete set $\boldsymbol{V}$. This requires that:

- (1) the assertions of $\mathbf{A}$ correspond exactly to undesirable behaviors: for any vector of values $\boldsymbol{v}$, an assertion in A is not satisfied if and only if $\diamondsuit\,(\boldsymbol{v})$ fails;
- (2 and 3) the precise statuses assigned to assertions in A are correct: the assertions bound to `true` (resp. `false`) in A are satisfied (resp. their negation is satisfied) for all values in $\boldsymbol{V}$;
- (4) if $\mathbf{A}$ is `closed`, the conjunction of all its assertions ensures the absence of undesirable behavior: if all these assertions are satisfied for values $\boldsymbol{v}$ in $\boldsymbol{V}$, then the computation of $\diamondsuit\,(\boldsymbol{v})$ succeeds.

This definition extends easily to any expression e and concrete state $\mathfrak{m}$. The evaluation $\overline{[\![e]\!]_{\mathfrak{m}}}$ fails if one of its operators fails. A map of alarms A is a sound abstraction of $\overline{[\![e]\!]_{\mathfrak{m}}}$ if each assertion of A prevents an undesirable behavior of an operator in e, if the statuses of A are correct in $\mathfrak{m}$, and if the map is `closed`, then the evaluation of e succeeds whenever all the assertions of the map are satisfied. This is denoted by $\mathbf{A} \models_{\mathbb{A}} \overline{[\![e]\!]_{\mathfrak{m}}}$.

The alarms are also equipped with a bounded lattice structure. The join $\sqcup_a$ and meet $\sqcap_a$ are defined pointwise. The domains of the two maps are equalized,

$$
\begin{array}{r|l}
\gamma_v : \mathbb{V}^\# \to 2^{\overline{\mathbb{V}}} & \forall \boldsymbol{e} \in e^n,\ \forall \boldsymbol{v} \in (\mathbb{V}^\#)^n,\ \forall r \in \mathbb{V}^\#, \\
\mathtt{F}^\#_\Diamond : e^n \to (\mathbb{V}^\#)^n \to \mathbb{V}^\# \times \mathbb{A} & \quad \begin{cases} \mathtt{F}^\#_\Diamond(\boldsymbol{e}, \boldsymbol{v}) = (r, \mathbf{A}) \Rightarrow \begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{[\![\Diamond(\boldsymbol{e})]\!]}(\gamma_v(\boldsymbol{v})) \\ \overline{[\![\Diamond(\boldsymbol{e})]\!]}(\gamma_v(\boldsymbol{v})) \setminus \varOmega \subseteq \gamma_v(r) \end{cases} \\ \{\boldsymbol{x} \in \gamma_v(\boldsymbol{v}) \mid \overline{[\![\Diamond(\boldsymbol{e})]\!]}(\boldsymbol{x}) \in \gamma_v(r)\} \subseteq \gamma_v(\mathtt{B}^\#_\Diamond(\boldsymbol{v}, r)) \end{cases} \\
\mathtt{B}^\#_\Diamond : (\mathbb{V}^\#)^n \times \mathbb{V}^\# \to (\mathbb{V}^\#)^n &
\end{array}
$$

Fig. 6. Semantics of values

by adding in each the assertions present only in the other, with a `true` status for `closed` maps and an `unknown` for `open` ones. Then, the join or meet of the statuses lattice is applied pointwise on the maps. The join with an `open` map returns an `open` map, while the meet with a `closed` map returns a `closed` map. The meet may discover an inconsistency between statuses, which stops the analysis. The bottom of the alarms lattice is the `closed` empty map, denoting an absence of undesirable behavior. Its top is the `open` empty map: any undesirable behavior may happen, and no assertion has a precise status.

4.2 Values

Values in $\mathbb{V}^\#$ are non-relational abstractions of sets of concrete values. They are equipped with a meet-semilattice structure, and provide a forward and a backward abstract counterpart $\mathtt{F}^\#_\Diamond$ and $\mathtt{B}^\#_\Diamond$ for each C operator $\Diamond$ on expressions. We define the soundness of the value abstractions through a concretization function γ_v that connects each value to the set of concrete values it represents. Then, the lattice must have the properties specified in Fig. 3, while the correctness of the abstract semantics is stated in Fig. 6.

Given value abstractions of the arguments, the forward abstract operator $\mathtt{F}^\#_\Diamond$ produces an alarm map and a value: the alarms are a sound abstraction of the undesirable behaviors of the operation, and the value is an over-approximation of the set of possible resulting C values when no undesirable behavior occurs. The forward operators receive the involved expressions needed to return the alarms.

Conversely, the backward operator $\mathtt{B}^\#_\Diamond$ tries to reduce the abstractions for its arguments, according to an abstraction of the result. The reduced abstractions over-approximate all the possible C values for the arguments leading to a value included in the result through the operator. For instance, the backward operator on interval abstractions for the comparison $\top \leq [0..10]$ with result $[1]$ reduces the first argument value to $[-\infty..10]$, and lets the second argument value unchanged.

Values are abstractions of C values of scalar types only. However, in a language such as C, variables may contain addresses. Hence, values must also be abstractions of memory locations, and their abstract transformers encompass all operations involving addresses and lvalues. Such abstractions allow the domains to express properties about memory locations, including pointer aliasing. Thus, the analyzer does not depend on a specific pointer analysis or memory model. Instead, these features are implemented as –potentially dedicated– domains that exchange their results with the others.

```
type eval = value * alarms
val F#_*τ : oracle:(exp -> eval) -> state -> location -> eval
val F#_𝔻  : oracle:(exp -> eval) -> state -> exp -> eval
```

$$\forall e \in expr,\ \forall v, l \in \mathbb{V}^{\#},\ \forall S \in \mathbb{D},\ \forall \mathbf{A} \in \mathbb{A},$$
$$\begin{cases} \mathtt{F}^{\#}_{*_\tau}(S,l) = (v,\mathbf{A}) \Rightarrow \forall a \in \gamma_v(l), \forall \mathfrak{m} \in \gamma_d(S), \begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{[\![*_\tau a]\!]}_{\mathfrak{m}} \\ \mathfrak{m}_\tau[a] \in \gamma_v(v) \cup \{\Omega\} \end{cases} \\ \mathtt{F}^{\#}_{\mathbb{D}}(S,e) = (v,\mathbf{A}) \Rightarrow \forall \mathfrak{m} \in \gamma_d(S), \begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{[\![e]\!]}_{\mathfrak{m}} \\ [\![e]\!]_{\mathfrak{m}} \in \gamma_v(v) \cup \{\Omega\} \end{cases} \end{cases}$$

Fig. 7. Interface and soundness of domains queries

5 State Abstraction

Abstract domains carry state abstractions collecting properties about program variables. For an abstract domain $\mathbb{D}$, a concretization γ_d links abstract states S to sets of concrete memories $\mathfrak{m}$. The soundness of an abstract domain is defined according to the concretization.

$$\gamma_d : \ \mathbb{D} \to \mathcal{P}(\mathfrak{M})$$

An abstract domain supplies the generic evaluator with value abstractions on lvalues, but also on whole expressions. When preparing its answer, it may also request additional information from other domains.

5.1 Domain Queries

A state abstract domain $\mathbb{D}$ must provide two *query* functions, on which the generic evaluator relies. Those functions extract information from an abstract state, translating AST fragments into alarms and value abstractions. Fig. 7 shows their signature, as well as their soundness requirements. The explanation of the oracle argument is postponed to the next subsection.

The first query is an abstract semantics $\mathtt{F}^{\#}_{*_\tau}$ for dereferences. It receives the possible memory locations of the dereferenced lvalue, and computes a sound value abstraction of the C values that may be stored in these locations, in all the memories abstracted by a state. It also produces an alarm map that ensures the validity of the location, and that the contents of the read memory slice are proper (i.e. not *indeterminate* in C parlance, and in particular initialized). Using $\mathtt{F}^{\#}_{\diamond}$ and $\mathtt{F}^{\#}_{*_\tau}$, an expression can be fully evaluated by induction on its syntax.

The second query $\mathtt{F}^{\#}_{\mathbb{D}}$ supplies additional information about arbitrary C expressions. It computes sound alarm maps and value abstractions for their evaluations in all the memories abstracted by a state. For instance, a domain tracking inequalities may express that $e_1 - e_2$ is positive when it has inferred $e_1 \geq e_2$. For any query on which the domain has no precise information, the top elements of values and alarms are always a sound over-approximation.

A generic evaluator queries the state domains on each lvalue and expression. If the domain is a combination of domains, all their answers are intersected using the $\sqcap$ operator on alarms and values. Thereby, each domain may easily contribute to reduce the abstract value computed for an expression, or decrease the number

```
1  int t[5] = {1, 2, 3, 4, 5};
2  int tmp = t[i]+1;
3  if (i == 2)
4      • r = t[i] + 1;
```

	Env	Array	Eq
State S	$i \mapsto [2]$ $tmp \mapsto [2..6]$	$t:$ $[1;2;3;4;5]$	$tmp = t\,[i] + 1$ $i = 2$
tmp	$[2..6]$	$\top$	$[\![*\,(\&t\,[i])\,{+}1]\!]^{\#}$
i	$[2]$	$\top$	$\top$
$t\,[i]$	$\top$	$[3]$	$\top$
$t\,[i] + 1$	$\top$	$\top$	$[\![tmp]\!]^{\#}$

$$[\![i]\!]^{\#}(S) = \mathtt{F}^{\#}_{*\mathtt{int}}\,(S, \&\mathtt{i}) = [2] \sqcap_v \top_v = [2] \tag{1}$$

$$[\![\&t\,[i]]\!]^{\#}(S) = (\&\mathtt{t}\to 0) +^{\#} \mathtt{sizeof}\,(\mathtt{int}) \times^{\#} [\![i]\!]^{\#}(S) = \&\mathtt{t}\to 8 \tag{2}$$

$$[\![*(\&t\,[i])]\!]^{\#}(S) = \mathtt{F}^{\#}_{*\mathtt{int}}(S, \&\mathtt{t} \to 8) = \top_v \sqcap_v [3] = [3] \tag{3}$$

$$[\![*(\&t\,[i]){+}1]\!]^{\#}(S) = [\![*(\&t\,[i])]\!]^{\#}(S)\ +^{\#}\ [\![1]\!]^{\#}(S) = [4] \tag{4}$$

Fig. 8. Collaboration between domains

of emitted alarms. If the values are themselves a combination of abstractions, each domain may have a precise answer for some value components and return $\top$ for the others. Using this lightweight collaboration mechanism, abstract domains may track a specific undesirable behavior, such as the initialization of variables. Open maps of alarms allow a domain to assert that some alarms cannot happen, without understanding e.g. the contents of variables. They also may collect properties only on a subset of the C language, and rely on the other abstractions to interpret together the whole semantics.

Example 2. Fig. 8 illustrates the collaboration between state abstractions, when analyzing the C code at the left, where the value of i ranges between 0 and 4 at the first line. We use intervals as arithmetic value abstractions and maps from memory bases to intervals-expressed byte offsets as pointer value abstractions. Two abstract domains cooperate, an environment mapping integer variables to intervals, and an array domain, able to represent precisely the value of each array cell. A third domain gathering symbolic equalities between expressions will be used later. The state S of the domains at the bullet point is given in the first line of the table, as well as their answers to some queries in the following lines; $[i]$ represents a singleton interval. In this example, each domain has information about some expressions, and returns $\top$ for the others.

We focus on the evaluation of expression $*\,(\&t[i]) + 1$ at line 4. We omit here the calls of the domain queries on non-lvalue expressions, as the domains have no information about them. We also write the abstract value semantics $\mathtt{F}^{\#}_{\Diamond}$ with an infix notation $\Diamond^{\#}$. The equations of Fig. 8 detail the steps of the evaluation, which proceeds bottom-up. First, for the variable i, the environment gives the precise value $[2]$, while the array domain returns $\top_v$. The meet of those two values is $[2]$. Then, the abstract value operator on array subscripts computes an abstraction for the address of `t[i]`, namely $\&\mathtt{t} \to [8]$. Third, using this precise abstraction of the address, the array domain is able to provide a precise value for the dereference of `t[i]`, which is $[3]$. Last, the abstract addition semantics applied on `t[i]` and 1 finally leads to $[4]$ as the value being assigned to r.

```
1 int t[4] = {1, 2, 3, 4};
2 int tmp = t[i]+1;
3 if (i == 2) • r = tmp;
```

$$\mathtt{F}^{\#}_{\mathtt{*int}}\,(S, \&tmp) = [2..6] \sqcap_v [\![t[i]{+}1]\!]^{\#} = [2..6] \sqcap_v [4] = [4]$$

Fig. 9. Using the oracle during evaluation

5.2 Interaction through the Oracle

To compute precise abstractions for an expression, a domain —and especially a relational one— may need additional information about other expressions. Thus, the domain can request the evaluation of new expressions, through the `oracle` argument of the query functions. The oracle triggers the requested evaluation using all available domains, and returns the cooperatively computed abstractions to the initial domain. The oracle has the same specification as the evaluation: in the current state S, it provides an alarm map and a value that are sound approximations of the concrete evaluation of the expression.

$$\mathtt{oracle}\,(S, e) = (v, \mathbf{A}) \Rightarrow \forall \mathfrak{m} \in \gamma_d\,(S)\,, \begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{[\![e]\!]_{\mathfrak{m}}} \\ [\![e]\!]_{\mathfrak{m}} \in \gamma_v\,(v) \cup \{\Omega\} \end{cases}$$

An uncontrolled use of the oracle may lead to (1) a loop in the forward evaluation, or (2) to an infinite chain of evaluations of different expressions. To prevent (1) from happening, the oracle can return $\top$ on re-occurrences of the same expression. The number of recursive uses of the oracle is also limited by a parameter of the analysis, to avoid (2).

Thanks to the oracle, the abstract domains share information through value abstractions, without a direct communication between domains. Especially, the oracle allows a relational domain to fully avail the relations it has inferred, and lets the other domains collaborate in leveraging these relations. The following example illustrates this with a simple equality domain.

Example 3. Let us come back to Example 2, but on the variant shown in Fig. 9 and with the equality domain enabled. At line 4, the value of t[i]+1 is still assigned to r, but through the intermediate variable *tmp*. The abstract states shown in Fig. 2 remain valid. Note that the environment domain maps *tmp* to the interval [2..6], coming from its assignment to t[i]+1 at line 2, when i was still imprecisely known; the value for t[i] was then provided by the array domain, and was not reduced by the condition at line 3.

However, the equality domain can use information it has inferred, namely tmp==t[i]+1. When the generic evaluator requests a value for *tmp*, the equality domain queries the value of t[i]+1 through the oracle. The main forward evaluation computes the precise interval [4] for t[i]+1, as in the initial example. This abstract value is finally returned by the equality domain as a sound value abstraction of *tmp*. Thanks to the equality domain, we obtain the same precision for the value assigned to r as in the original example.

During the evaluation of t[i]+1, the equality domain may request the evaluation of *tmp* through the oracle. Then, the evaluator detects a loop in the evaluation, and returns the top abstractions without any further computation.

```
val B#_{*τ} : state -> location -> value -> location
val B#_𝔻   : state -> exp -> value -> (exp * value) list
```

$$\forall e \in \mathit{expr}, \forall v, l \in \mathbb{V}^{\#}, \forall S \in \mathbb{D},$$

$$\begin{cases} \{a \in \gamma_v(l) \mid \exists \mathfrak{m} \in \gamma_d(S), \mathfrak{m}_\tau[a] \in \gamma_v(v)\} \subseteq \gamma_v(\mathtt{B}^{\#}_{*_\tau}(S, l, v)) \\ \forall (e', v') \in \mathtt{B}^{\#}_{\mathbb{D}}(S, e, v),\ \forall \mathfrak{m} \in \gamma_d(S),\ \overline{[\![e]\!]}_{\mathfrak{m}} \in \gamma_v(v) \Rightarrow \overline{[\![e']\!]}_{\mathfrak{m}} \in \gamma_v(v') \end{cases}$$

Fig. 10. Backward propagation inside domains

5.3 State Backward Propagation

Within the value abstraction, forward and backward propagators are dual. Likewise, abstract domains must provide the backward counterparts of queries. Fig. 10 presents their requirements. When the abstract value $v^{\#}$ stored in a lvalue is reduced, the abstract memory location $l^{\#}$ for the lvalue might be reduced as well (e.g. when some locations of $l^{\#}$ are known not to contain $v^{\#}$). This typically happens on memory accesses through an imprecise pointer. The backward semantics $\mathtt{B}^{\#}_{*_\tau}$ of dereference serves this purpose: it takes an abstract state, an abstraction of the memory location of a lvalue and its new value abstraction v. It returns a possibly more precise location abstraction, which is an over-approximation of the concrete locations for which the lvalue has a value in $\gamma_v(v)$.

Moreover, the relations known by a relational domain may also induce further interesting reductions. For instance, if $a \leq b$ holds, then any reduction of the infimum of the possible values of a implies the same reduction for b. Hence, when performing a reduction, the generic evaluator notifies the domains through the function $\mathtt{B}^{\#}_{\mathbb{D}}$, which returns a list of new reductions to be backward propagated by the evaluator. The new reductions, deduced from the prior one and from the inferences made by the domain, must be correct in the concrete states for which the initial reduction was valid. To avoid diverging, the generic evaluator must limit the number of times this function is used.

5.4 Abstraction of Statement Semantics

The generic transfer function on statements starts by evaluating all involved expressions. For an assignment, the location of the lvalue is also evaluated, and reduced to its valid part. The alarms produced at each step are accumulated, and eventually raised to report all undesirable behaviors that may have occurred at this point. The concrete states satisfying the emitted assertions are ensured to succeed on this statement. Then, the abstract transformers of the domains are applied. They have to be a sound approximation of the program semantics for these safe states. For that purpose, each domain can use the value abstractions that have been cooperatively computed by the evaluator. Thus, an abstract transformer benefits from the properties inferred by all domains.

6 EVA: a Modular Abstract Interpreter for Frama-C

We have implemented the architecture described in this paper in an extensible, modular abstract interpreter named EVA, an open-source plugin of FRAMA-C[3]. EVA handles the subset of C99 commonly used in embedded code, as well as some extensions[4]. It detects the most common undefined behaviors of the C standard [14], including invalid memory accesses, reading uninitialized memory, divisions by zero, integer overflows, undefined bit shifts, writes in `const` memory, reads of bits of a dangling address, invalid pointer comparisons and subtractions, infinite or NaN floating-point values[5].

This section presents the value and state abstractions currently available in EVA. In particular, the *Cvalue* domain implements the abstract semantics of VALUE, the former abstract interpreter of FRAMA-C. The five other abstract domains are new. By lack of space, we only give a short overview.

EVA provides several *value abstractions* (and their semantics) establishing an already rich communication interface between abstract domains. These abstractions may be extended to achieve further communication. The current integer abstractions are a reduced product between small sets of discrete integers (whose maximal cardinal is user-configurable), integer intervals and linear congruences. Floating-point abstractions are intervals, excluding infinite and NaNs. Pointer and location abstractions are maps from memory bases (roughly, variables) to byte offsets represented by an integer value. Pointer values may thus express precise alias information between program variables. Such alias information is especially useful for numerical domains that do not include an alias analysis, in particular to process assignments through pointers. Numeric domains may also collaborate to reduce the possible offsets on a variable. The operators for these value abstractions handle all kinds of alarms, and always produce closed maps of alarms. New value abstractions are thus simpler to write: they may limit themselves to a subset of C and focus on a certain kind of alarms (through `open` maps of alarms).

The *Cvalue domain* is the biggest abstract domain of EVA. It is inherited from VALUE, and was retrofitted for EVA. It uses the standard values of EVA. Its state domain is quite involved, and we refer the reader to [17,3] for a more complete explanation. The memory is (roughly) a map from *variable* $\times$ *offset* $\times$ *width* to abstract values, plus two additional booleans that abstract the possibility that the value may be uninitialized, or a dangling pointer. The memory is untyped, and it is possible to write an abstract value of any type anywhere in the memory. Assignments overlapping existing bindings are automatically handled, and

[3] Directory `src/plugins/value/` of the FRAMA-C source files, available at `http://frama-c.com/download.html`

[4] Bitfields, flexible array members and some GNU extensions are supported. Support for dynamic allocation is preliminary. Recursion, `setjmp/longjmp`, `complex` types, `alloca` and variable-length arrays are not supported.

[5] These are not undefined behaviors w.r.t. the ISO C99 or IEEE 754 specifications, but we choose to report them as undesirable errors.

remain precise. Assignments to a very large number of non-contiguous locations are automatically approximated.

The *equality domain* is a symbolic domain tracking Herbrand equalities between C expressions. Our intentions are somewhat similar to those of Miné [21], in particular abstracting over temporary variables resulting from code normalization. The equalities are deduced from equality conditions and from assignments. The (cooperatively computed) information about locations are used to invalidate equalities that may no longer hold after an assignment. This domain uses the oracle and its backward counterpart (Section 5.3) to avail its inferred relations. It is thus independent of the chosen value abstraction, and is implemented by a functor from values to state abstractions.

The *symbolic locations domain* tracks accesses to arrays or through pointers in a symbolic way. It intends to precisely analyze codes such as `if (t[i]<e) v=t[i]`. Indeed, when i is imprecise, domains that represent arrays in extenso cannot learn information from the condition (because any cell may be involved). The domain shares some similarities with the *recency* abstraction [1]. Its state is a map from symbolic locations (such as `t[i]`, `*p` or `p−>v`) to an abstract value. Strong reductions are performed on those values when analyzing conditions, to be shared with the other domains when the location is encountered again later.

The *Apron domains*: we have implemented a simple binding to the numerical abstract domains available in APRON [15]. The resulting domains (boxes, octagons, strict or loose convex polyhedra, linear equalities) demonstrate that the relational domains of Apron fit easily within the communication model of EVA. The abstract state is an APRON state. Since those contain no aliasing information, the binding relies instead on the other domains (mostly CVALUE) to evaluate memory locations. A mapping between the APRON dimensions and the variables of the program is used as a correspondence table. The domain answers queries for arithmetic expressions, by translating them into the APRON internal language. Sub-expressions that cannot be handled by APRON are linearized on-the-fly into intervals, using the cooperatively computed value.

The *bitwise domain* aims at adding bitvector-like reasoning to EVA (including on floating-point values and pointers), without resorting to a dedicated implementation. Instead, we reuse the expressivity of the abstraction for sequences of bits in the CVALUE domain. Indeed, this abstraction is already able to extract the possible values of some bits in a memory range. This bitwise domain works on a new kind of value abstractions, namely a sequence of bits of known length. Only the forward and backward abstract semantics for the bitwise C operators, as well as integer casts and multiplication/division by a power of 2, have been implemented. All other operations degenerate to $\top_v$. The reduced product between the standard values and those new bitwise values performs a conversion between the two representations when possible.

The *gauges domain* [22] is a weakly relational domain, able to efficiently infer general linear inequality invariants within loops. Technically, the variables involved in the invariants are all related to *loopcounters*, that model the current number of iterations in each loop. Gauges are especially useful to infer invariants for pointer offsets, as pointer arithmetic introduces +4 or +8 increments (for 32- and 64-bits architecture respectively), that cannot be directly handled by domains such as octagons [19]. The gauge domains communicates integer and pointer values through the standard values of EVA.

We believe the variety of domains presented above validates our design choices on how to structure a collaborative analyzer. Having an implementation of values independent from domains is natural, and avoids code duplication. The cooperative evaluation of value abstractions achieves an exchange of information between abstract domains, without direct interactions. This modularity facilitates the introduction of new abstractions. Furthermore, allowing the relational domains to directly trigger new complete evaluations or backward propagations spares the other domains from processing relational instructions. Finally, having only value and location abstractions as a direct means of communication did not feel limiting, especially since they are extensible.

7 Related Works

Splitting value abstractions from state abstractions was proposed by Cousot [8] to design an abstract interpreter, but was not used to enable a communication between different state abstractions. Cortesi *et al.* [7] survey the use of products (reduced or not) in abstract interpretation. Although most abstract-interpretation-based analysis frameworks use multiple domains internally (e.g. [5]), few explain how the different domains exchange information. In theory, a reduced product considers the equivalence classes of the direct products that have the same concretization, and reduces the result of each abstract operation to the smallest representative of its class. Abstract interpreters usually implement an approximation of the reduced product, but let the domains interact during an operation.

The *open product* [6] achieves the reduction by a set of boolean functions (*queries*) provided by any domain and used by the abstract operators to receive more information from the environment. Our design defines clearly the scope of our queries (through value abstractions), and goes beyond direct queries between domains by sharing all the evaluation engine of expressions to facilitate the interpretation of a statement: abstractions of expressions are automatically computed from properties expressed on subterms (and conversely through backward propagation).

Astrée implements an approximate reduced product through *communication channels* [11]. This mechanism has later been implemented in Clousot and Verasco [13,16]. Each channel carries an information of a certain kind: interval range, integer congruence, equalities between expressions, etc. New messages can be added if needed, and domains need not to understand all messages. Messages sent on channels play a role similar to our value abstractions, but important differences exist. First, our design allows domains to cooperate at another level, namely by emitting statuses on alarms. Second, maintaining a network of communication channels in parallel of all transfer functions seems more invasive (from an engineering point of view) than using value abstractions. Indeed, the latter are naturally understood by the evaluation functions. Third, our oracle mitigates the need for messages containing relational information, that must be understood and processed by non-relational domains. With the oracle, no new abstract transformer needs to be added. Fourth, the reduced product of value abstractions allows information to flow between domains, even if they understand different values. Thus, domains need not be adapted when a new kind of value is added. Finally, our products are unordered, while communication channels are oriented. This potentially allows for more reduction opportunities in the domains.

Beyer *et al.* [2] propose an extension of *configurable program analysis* (CPA) in which a *precision* information is tracked. This precision is used to dynamically alter the amount of information the abstract domain infers. The composition of two domains is done through a cartesian product, except that the functions related to precision can use information from both domains. This way, it is possible to reduce the precision of a domain when another one is precise enough.

The CodePeer analyzer [4] for Ada uses internally an SSA form on value-numbered expressions to represent its abstract state. A state is a mapping from SSA expressions to an abstract value. (The domains for values include disjunctions of integer intervals and floating-point intervals.) Storing information for entire expressions alleviates the need for relational domains; instead, a value for e.g. $x - y$ is stored. There are some similarities between this state and the partial maps internally used by the EVA evaluator to store the value abstractions of expressions. However, a major difference is that we reset the map after each statement. Keeping such information longer could be useful.

8 Conclusion

EVA is a major development: 13k lines of new or heavily adapted code, out of 53k for EVA, VALUE and all the shared abstractions. It has already replaced VALUE as the default abstract interpreter in the latest version of FRAMA-C. The new domains we have implemented validate our design so far. The separation between value and state abstractions is conceptually useful, and reduce the amount of code that must be written when new state domains are implemented. We plan to mature the domains presented in Section 6, and to write new value abstractions (e.g. to represent structs) and state abstractions (e.g. to improve the handling of dynamic allocation).

Regarding collaborative evaluation, two important things remain to be done. First, VALUE uses an automatic summarization mechanism to speed up analyses [23]. It needs to be extended to arbitrary domains, while remaining cost-efficient. Second, in the current implementation of EVA, domains cooperate only to evaluate the C part of the AST. A collaborative evaluation of logical assertions is a worthwhile goal. However, this will probably complexify the abstract values and domains, that will have to understand the fine print of assertions (e.g. real numbers in specifications).

References

1. Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.
2. Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *ASE*, pages 29–38, 2008.
3. Richard Bonichon and Pascal Cuoq. A mergeable interval map. *Stud. Inform. Univ.*, 9(1):5–37, 2011.
4. Jean-Louis Boulanger, editor. *Static Analysis of Software: The Abstract Interpretation.* Wiley-ISTE, 2011.
5. Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods*, pages 271–277, 2014.
6. Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Sci. Comput. Program.*, 38(1-3):27–71, 2000.
7. Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. A survey on product operators in abstract interpretation. In *Essays Dedicated to D. Schmidt on the Occasion of his 60th Birthday*, pages 325–336, 2013.
8. Patrick Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design.* NATO ASI Series F. IOS Press, 1999.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles Of Programming Languages*, pages 238–252, 1977.
10. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles Of Programming Languages*, pages 269–282, 1979.
11. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the Astrée static analyzer. In *Advances in Computer Science - ASIAN*, pages 272–300, 2006.
12. Pascal Cuoq, Philippe Hilsenkopf, Florent Kirchner, Sébastien Labbé, Nguyen Thuy, and Boris Yakobowski. Formal verification of software important to safety using the Frama-C tool suite. In *NPIC & HMIT*, 2012.
13. Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verif. of Obj.-Oriented Software*, pages 10–30, 2010.
14. International Organization for Standardization (ISO). *International Standard ISO/IEC 9899:1999 - Programming languages - C*, 2007. Technical Corrigendum 3.
15. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667, 2009.
16. Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Princ. Of Prog. Lang.*, pages 247–259, 2015.
17. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
18. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *Europ. Symp. on Programming*, pages 5–20, 2005.
19. Antoine Miné. The octagon abstract domain. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings of the Eighth Working Conference on Reverse*

Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001, page 310. IEEE Computer Society, 2001.

20. Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.
21. Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.
22. Arnaud Venet. The gauge domain: Scalable analysis of linear inequality invariants. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2012.
23. Boris Yakobowski. Fast whole-program verification using on-the-fly summarization. In *Workshop on Tools for Automatic Program Analysis*, 2015.