Under-approximating Memory Abstractions

Marco Milanese, Antoine Miné

Under-approximating Memory Abstractions

Marco Milanese[1][0000−0002−6215−7359] and Antoine Miné[1][0000−0002−6375−3179]

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
firstname.lastname@lip6.fr

Abstract. This work presents a sound backward analysis for sufficient pre-conditions inference in C programs, by abstract interpretation. It utilizes the under-approximation abstract operators studied by Milanese and Miné [35,32] for a purely numeric subset of C, and extends them to support the C memory model. Pointer dereferences are handled with the cell abstraction [34] and dynamic memory allocations with the recency abstraction [3]. A direct usage of the abstract operators proposed in these previous works in an under-approximation framework is not possible as either internally they rely on over-approximated operators (e.g., cell removal) or an extension to this framework is not straightforward (e.g., under-approximating join). In this work we propose new operators that are under-approximating, and on top of this we design a backward semantics. The analysis is implemented in the MOPSA analyzer and its performance is assessed in detection of sufficient pre-conditions for runtime errors in 13,261 C tasks from NIST Juliet test suite [5].

Keywords: Abstract Interpretation · Software Verification · Program Analysis · Bug Catching · Under-approximation.

1 Introduction

Traditionally, static analyses based on abstract interpretation [13,14,15] focused on computing over-approximations of the concrete semantics of programs. This allows to formally *certify* that programs satisfy some specification but, by the virtue of the analysis, can introduce false positives. This may not be acceptable in contexts where static analysis tools are deployed to help developers *catching* bugs. In these contexts it is more important to raise only true alarms than to raise alarms for all possible bug conditions. For this purpose, we need an *under-approximating* rather than over-approximating analysis. Additionally, to maximize the information content provided to developers, the analyzer should

```c
union num {
    int i;
    float f;
};
union num *p = malloc(sizeof(union num));
p->i = 0;
assert(p->f == 0);
```

Fig. 1: Simple C program requiring a precise representation of memory.

provide input cases, possibly as simple as possible, which can trigger the bug. For instance in the following C program

```c
int arr[10];
arr[getchar()] = 0;
```

the analysis tool should report that the array access is unsafe, and more specifically that it is sufficient for `getchar()` to return a value greater than 9 for the access to be unsafe.

The authors of [32,35] studied a backward analysis based on abstract interpretation allowing to infer *sufficient pre-conditions*. This class of pre-conditions differs from classic necessary pre-conditions in the handling of non-determinism. For a post-condition S, sufficient pre-conditions contain states σ such that $\forall \sigma'. (\sigma \to \sigma') \implies \sigma' \in S$ whereas necessary pre-conditions contain states σ such that $\exists \sigma' \in S.\ \sigma \to \sigma'$, where we model the program as a transition system and $\to$ denotes the transition relation. In other words, sufficient pre-conditions ensure that no matter the non-deterministic trajectory of the program, if the program terminates, it terminates in a state satisfying the post-condition. Vice versa, necessary pre-conditions ensure that there exists a program trajectory which terminates in the post-condition. Because of this property, sufficient pre-conditions are more appealing in a bug catching context as they allow finding bugs that occur for *any* program run.

Motivation. Previous work on sufficient pre-condition computation by abstract interpretation focuses on programs with only numeric types. Unfortunately this falls short of the analysis of C programs, where non-scalar types, memory allocations, and pointers are commonly used. Moreover, in C, due to type punning, it is possible to access to the same memory location with different types. For example the code of Fig. 1 accesses the same memory location with fields `i` and `f`. In order to handle these features, abstractions beyond numeric domains are required. In this work we aim at filling this gap by studying how well-known abstractions for pointers, namely the cell abstraction [34], and dynamic memory allocations, namely heap-recency abstraction [3] (at allocation sites), can be employed in an under-approximation sufficient pre-conditions analysis.

Contribution. The backward analysis considered in this work requires under-approximated operators. This poses several challenges, especially in the cell abstraction, as some operators, namely join and cell removal, can over-approximate the state. In particular, this occurs in the join because the cell domain is constructed as a Cartesian product of subdomains, and the usual construction of the join as a point-wise lift of the joins of the subdomains can yield spurious states. Consider for instance under-approximating the join of pointers $\&a{+}1$ and $\&b{+}2$: handling separately the join of variables $\{\&a\}\cup\{\&b\}$ and offsets $\{1\}\cup\{2\}$ leads to $\{\&a, \&b\}{+}[1, 2]$, which is not a sound under-approximation of $\{\&a{+}1\}\cup\{\&b{+}2\}$, although $\{\&a, \&b\}$ and $[1, 2]$ are valid under-approximations of $\{\&a\}\cup\{\&b\}$ and $\{1\} \cup \{2\}$ respectively as they represent the exact join. In general, the Cartesian product of over-approximations is an over-approximation, but the Cartesian product of under-approximations may not be an under-approximation. To overcome this difficulty, we study a special join which under-approximates its arguments so that the point-wise construction can be used while still retaining an under-approximation of the concrete union. Similarly, the cell removal operator can introduce over-approximations as it removes cells by simply dropping them from the state. In the cell domain, cells act as constraints on the invariant: removing them *enlarges* the set of represented states, and thus over-approximates. To overcome this limitation, we show how the information contained in the cell to be removed can be first transferred to other cells, so that the cell can be later removed safely.

In summary, the contribution of this work is threefold:

- In Sect. 2, we extend the cell abstraction with under-approximated lattice operators and design a backward semantics based on this domain which can handle the C semantics of pointers and memory;
- In Sect. 3, we design a backward semantics for memory allocation and free operators based on the heap-recency abstraction;
- In Sect. 4, we implement this backward analysis in the MOPSA [26] static analysis platform and evaluate experimentally the accuracy of the analysis by analyzing 13,261 C tasks from the NIST Juliet collection of benchmarks [5].

Section 5 presents related works and Section 6 concludes.

2 Memory Abstractions

Previous works on sufficient pre-conditions inference by abstract interpretation such as [32,35] were limited to purely numeric semantics. Here, we focus on extensions of this semantics to handle memory and non-scalar types of the C programming language, i.e., aggregate types (e.g., structs, unions, arrays), pointers and casts. This section is organized as follows: Sect. 2.1 recalls a low-level semantics of memory, Sect. 2.2 recalls the cell abstraction and its semantics and Sect. 2.3 recalls the abstract semantics with cells, Sect. 2.4 focuses on a backward semantics for the cell concrete semantics, Sect. 2.5 focuses on the design of under-approximated lattice operators for the cell abstract domain and Sect. 2.6 focuses on a backward semantics for the cell abstract domain.

$$
\begin{array}{rll}
\text{int-sign} ::= & \texttt{unsigned} \mid \texttt{signed} & \\
\text{int-size} ::= & \texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long} & \\
\text{int-type} ::= & \text{int-sign int-size} & \\
\text{float-type} ::= & \texttt{float} \mid \texttt{double} & \\
\text{scalar-type} \ni \tau ::= & \text{int-type} \mid \text{float-type} \mid \texttt{ptr} & \\
\text{Expr} \ni a ::= & [v_1, v_2] & v_1, v_2 \in \mathbb{R} \\
\mid & V & V \in \mathbb{V} \\
\mid & \&V & V \in \mathbb{V} \\
\mid & *_\tau\, a & \\
\mid & \circ\, a & \circ \in \{-, !, \ldots\} \\
\mid & a_1 \diamond a_2 & \diamond \in \{+, \leq, \&, \ldots\} \\
\text{Stat} \ni s ::= & *_\tau\, p := a \mid \textbf{assume}(a) & p \in \mathbb{V}_p
\end{array}
$$

Fig. 2: Language Syntax.

2.1 Low-level Semantics

In this section we recall the semantics studied in [34].

Notation. Given a set S, then $\mathcal{P}(S)$ denotes its power-set, that is the collection of all subsets of S. We denote with $f : X \rightharpoonup Y$ a partial function f, where elements of a domain X *may* be mapped to a co-domain Y. If ρ denotes a state (or environment), then $\rho(v)$ indicates the lookup of the variable v in ρ, and $\rho[v \mapsto x]$ indicates the replacement of the content of the variable v with the value x.[1]

Language. Let $\mathbb{V}$ be a set of (typed) variables and $\mathbb{V}_p \subseteq \mathbb{V}$ the subset of variables with pointer type. We support a simple C-like language with scalar types, i.e., numeric types and pointer types, and non-scalar types, i.e., arrays, structs, unions. For simplicity, we assume that all memory accesses to non-scalar types are translated into scalar accesses, therefore in our formalization we only focus on the latter. In Fig. 2 we report the syntax of the language. Scalar types can be either numeric types (integers and floats) or pointer types. Expressions can be either atomic or compositions of other expressions. Finally, we provide two atomic statements: assignments and boolean filters. For simplicity, the left-hand side of assignments is a dereference of a pointer variable, but it is easy to generalize this to a generic expression (as we do in the implementation). For the sake of brevity we omit inductive statements such as if-then-else blocks, statement concatenation and while loops, but they are supported and their semantics is the same as in [35].

[1] More formally: $\rho[v \mapsto x] \triangleq \lambda w. \begin{cases} x & \text{if } v = w \\ \rho(w) & \text{otherwise} \end{cases}$

$$
\begin{aligned}
E_b[\![[v_1, v_2]]\!]\rho &\triangleq \{v \mid v_1 \leq v \leq v_2\} \\
E_b[\![V]\!]\rho &\triangleq \{v \mid v \in \mathrm{bdec}_{\mathrm{typeof}(V)}\langle \rho\langle V, 0\rangle, \ldots, \rho\langle V, \mathrm{sizeof}(V) - 1\rangle\rangle\} \\
E_b[\![\&V]\!]\rho &\triangleq \{\langle V, 0\rangle\} \\
E_b[\![*_\tau a]\!]\rho &\triangleq \{v \mid \exists \langle V, o\rangle \in E_b[\![a]\!]\rho.\ \text{safe-access}_\tau(\langle V, o\rangle), \\
&\qquad v \in \mathrm{bdec}_\tau\langle \rho\langle V, o\rangle, \ldots, \rho\langle V, o + \mathrm{sizeof}(\tau) - 1\rangle\rangle\} \\
E_b[\![\circ\, a]\!]\rho &\triangleq \{\circ\, v \mid v \in E_b[\![a]\!]\rho\} \\
E_b[\![a_1 \diamond a_2]\!]\rho &\triangleq \{v_1 \diamond v_2 \mid v_1 \in E_b[\![a_1]\!]\rho, v_2 \in E_b[\![a_2]\!]\rho\} \\
\tau_b[\![*_\tau p := a]\!]\rho &\triangleq \{\rho[\langle V, o\rangle \mapsto b_0, \ldots, \langle V, o + \mathrm{sizeof}(\tau) - 1\rangle \mapsto b_{\mathrm{sizeof}(\tau)-1}] \mid \\
&\qquad \langle V, o\rangle \in E_b[\![p]\!]\rho, \text{safe-access}_\tau(\langle V, o\rangle), \\
&\qquad \exists v \in E_b[\![a]\!]\rho.\ \langle b_0, \ldots, b_{\mathrm{sizeof}(\tau)-1}\rangle \in \mathrm{benc}_\tau(v)\} \\
\tau_b[\![\mathbf{assume}(a)]\!]\rho &\triangleq \{\rho \mid \exists v \in E_b[\![a]\!]\rho.\ (v \in \mathbb{R} \implies v \neq 0) \wedge (v \in \mathrm{Ptr} \implies v \neq \texttt{NULL})\}
\end{aligned}
$$

where $\text{safe-access}_\tau(p) \triangleq \exists \langle V, o\rangle = p.\ \forall\, i < \mathrm{sizeof}(\tau).\ \langle V, o + i\rangle \in \mathrm{Addr}$.

Fig. 3: Byte-level semantics.

Byte-level Semantics. We start from a low-level semantics where memory is represented at byte level and accessed via pointers. Pointers span in a domain $\mathrm{Ptr} \triangleq (\mathbb{V} \times \mathbb{N}) \cup \{\texttt{NULL}, \texttt{invalid}\}$ where $\langle V, i\rangle$ represents the *ith* byte of the variable V, `invalid` represents a pointer that does not point to a variable (obtained for example from casting a non-zero number) and `NULL` represents a special kind of invalid pointer obtained from the cast of 0. We denote by Addr the set of addressable pointers, that is $\{\langle V, i\rangle \mid V \in \mathbb{V}, i < \mathrm{sizeof}(V)\}$, where $\mathrm{sizeof}(\tau)$ denotes the size of the type τ and $\mathrm{sizeof}(V)$ denotes the size of the type of V. States contain *byte-values* from a domain $\mathbb{B} \triangleq [0, 255] \cup (\mathrm{Ptr} \times \mathbb{N})$, hence they can be either a byte or a pair $\langle p, i\rangle \in \mathrm{Ptr} \times \mathbb{N}$ which is a symbolic value denoting the *ith* byte in the memory representation of the pointer value p (a symbolic encoding is needed because the precise encoding depends on the system, not the program). Therefore, program states are in $\mathcal{E}_b \triangleq \mathrm{Addr} \to \mathbb{B}$.

The semantics of expressions is a function $E_b[\![a]\!] : \mathcal{E}_b \to \mathcal{P}(\mathbb{I})$ where $\mathbb{I} \triangleq \mathbb{R} \cup \mathrm{Ptr}$. The semantics of statements is a function $\tau_b[\![s]\!] : \mathcal{E}_b \to \mathcal{P}(\mathcal{E}_b)$. As expressions return values in $\mathbb{I}$ and states store byte-values we need a way to convert between the two formats. The conversion is ABI-specific (e.g., due to endianness) and can be non-deterministic to account for invalid cases (e.g., decoding a pointer with integer type). For illustration purposes, without loss of generality, we assume the ABI of Intel x86 CPUs. Formally, the semantics is parametric with respect to a value encoding function $\mathrm{benc}_\tau : \mathbb{I} \to \mathcal{P}(\mathbb{B}^*)$ and a value decoding function $\mathrm{bdec}_\tau : \mathbb{B}^* \to \mathcal{P}(\mathbb{I})$. See [34] for further details. Value operators $\circ$ and $\diamond$ are defined according to their C semantics.[2] $\mathrm{typeof}(V)$ denotes the type of the variable V.

[2] Pointer arithmetic is supported too. For instance $\langle V, o\rangle + i = \langle V, o + i\rangle$.

$$
\begin{aligned}
E_c[\![[v_1, v_2]]\!]\sigma &\triangleq \{\langle \sigma, v\rangle \mid v_1 \leq v \leq v_2\} \\
E_c[\![V]\!]\sigma &\triangleq \{\langle\langle C', r'\rangle, r'(\langle V, 0, \text{typeof}(V)\rangle)\rangle \mid \\
&\qquad \langle C', r'\rangle \in \text{add-cell}(\langle V, 0, \text{typeof}(V)\rangle, \sigma)\} \\
E_c[\![\&V]\!]\sigma &\triangleq \{\langle \sigma, \langle V, 0\rangle\rangle\} \\
E_c[\![*_\tau a]\!]\sigma &\triangleq \{\langle\langle C'', r''\rangle, r''(\langle V, o, \tau\rangle)\rangle \mid \\
&\qquad \exists \sigma'.\ \langle \sigma', \langle V, o\rangle\rangle \in E_c[\![a]\!]\sigma \wedge \text{safe-access}_\tau(\langle V, o\rangle), \\
&\qquad \langle C'', r''\rangle \in \text{add-cell}(\langle V, o, \tau\rangle, \sigma')\} \\
E_c[\![\circ\, a]\!]\sigma &\triangleq \{\langle \sigma', \circ\, v\rangle \mid \langle \sigma', v\rangle \in E_c[\![a]\!]\sigma\} \\
E_c[\![a_1 \diamond a_2]\!]\sigma &\triangleq \{\langle \sigma'', v_1 \diamond v_2\rangle \mid \exists \sigma'.\ \langle \sigma', v_1\rangle \in E_c[\![a_1]\!]\sigma, \langle \sigma'', v_2\rangle \in E_c[\![a_2]\!]\sigma'\} \\
\tau_c[\![*_\tau p := a]\!]\sigma &\triangleq \{\sigma'''' \mid \exists \sigma'.\ \langle \sigma', v\rangle \in E_c[\![a]\!]\sigma, \\
&\qquad \exists \langle\langle C'', r''\rangle, \langle V, o\rangle\rangle \in E_c[\![p]\!]\sigma'.\ \text{safe-access}_\tau(\langle V, o\rangle), \\
&\qquad c = \langle V, o, \tau\rangle, \sigma''' = \langle C'' \cup \{c\}, r''[c \mapsto v]\rangle, \\
&\qquad \sigma'''' \in \text{remove-overlapping-cells}(c, \sigma''')\} \\
\tau_c[\![\textbf{assume}(a)]\!]\sigma &\triangleq \{\sigma' \mid \exists v.\ \langle \sigma', v\rangle \in E_c[\![a]\!]\sigma, \\
&\qquad (v \in \mathbb{R} \implies v \neq 0) \wedge (v \in \text{Ptr} \implies v \neq \texttt{NULL})\}
\end{aligned}
$$

Fig. 4: Cell concrete semantics.

For the sake of brevity, we skip error handling from the presentation of the semantics. The complete semantics would include an error state ω, reached by illegal operations (e.g., dereferencing a non-addressable pointer). The (simplified) semantics is reported in Fig. 3.

2.2 Cell-based Semantics

Byte-level semantics is not a convenient starting point for designing an abstract semantics, because an abstract semantics based on it would have to require frequent conversions between values and byte-based representations. This requires more expressive (and costly) domains to represent even simple properties of multibyte integers (e.g., the range of a 32-bit integer is expressed as a linear inequality relation on its bytes).

To sidestep these limitations [34] proposes a less concrete semantics where program states represent memory not directly byte-by-byte, but instead by tracking the value contained in some multibyte memory *cells*. A cell represents a memory region containing a scalar value, together with its type (specifying the encoding of the value). Formally, it is a triple $\langle V, o, \tau\rangle$ where $V \in \mathbb{V}$ is a variable, $o \in \mathbb{N}$ is the cell's offset, so that the base of the cell is given by the address of V plus the offset o and $\tau \in$ scalar-type is the type used to access the cell (which defines also its size). The universe of valid cells is $\text{Cells} \triangleq \{\langle V, o, \tau\rangle \mid V \in \mathbb{V}, o \in \mathbb{N}, \tau \in \text{scalar-type}, o + \text{sizeof}(\tau) \leq \text{sizeof}(V)\}$ and Cells_{ptr} denotes cells with

pointer type. Program states are pairs $\langle C, r\rangle$ where C is a set of cells and r an environment with domain C. More formally $\mathcal{E}_c \triangleq \{\langle C, r\rangle \mid C \subseteq \text{Cells}, r \in C \to \mathbb{I}\}$. If an address is covered simultaneously by multiple cells, its byte values are computed as the intersection of the byte values contributed by each cell. The state $\langle C, r\rangle \in \mathcal{E}_c$ is mapped by the concretization function γ_{Cell} to the set of byte-level states $\gamma_{Cell}\langle C, r\rangle \triangleq \{\rho \in \text{Addr} \to \mathbb{B} \mid \forall\langle V, o, \tau\rangle \in C.\ \exists\langle b_0, \ldots, b_{\text{sizeof}(\tau)-1}\rangle \in \text{benc}_\tau(r\langle V, o, \tau\rangle).\ \forall i < \text{sizeof}(\tau).\ \rho\langle V, o+i\rangle = b_i\}$. We denote with $\gamma_{Cell}\langle C, r\rangle(c)$ the set of byte strings in the memory region $c = \langle V, o, \tau\rangle$ in the state $\langle C, r\rangle$, that is $\{\langle b_0, \ldots, b_{\text{sizeof}(\tau)-1}\rangle \mid \exists\rho \in \gamma_{Cell}\langle C, r\rangle.\ \forall i < \text{sizeof}(\tau).\ b_i = \rho\langle V, o+i\rangle\}$.

Example 1. Let $c_1 \triangleq \langle V, 0, \texttt{unsigned int}\rangle$ and $c_2 \triangleq \langle V, 0, \texttt{unsigned char}\rangle$ be two overlapping cells. Consider a state $\sigma_1 \triangleq \langle\{c_1, c_2\}, [c_1 \mapsto 0, c_2 \mapsto 0]\rangle$. The byte-level concretization of σ_1 is a singleton byte-memory: $\{[\langle V, 0\rangle \mapsto 0, \langle V, 1\rangle \mapsto 0, \langle V, 2\rangle \mapsto 0, \langle V, 3\rangle \mapsto 0]\}$. However, the concretization of the state $\sigma_2 \triangleq \langle\{c_1, c_2\}, [c_1 \mapsto 0, c_2 \mapsto 1]\rangle$ is $\varnothing$ as the two cells have different values on the overlapping part. □

In order to avoid the encoding and decoding of values from and into bytes, this semantics implements memory accesses by reading and writing values to cells at the location of the memory access. However, since it is not possible to know in advance the set of cells that will be used during the analysis (e.g., due to type-punning), cells must be added dynamically, during the analysis, on-demand. In particular, the analysis starts with an empty set of cells, and then if a memory access targets a cell that is missing from the state, the cell is added so that its value can be retrieved or updated. Because of this, if the same part of memory is read with different types, overlapping cells can be created. This configuration requires special attention if a portion of the overlapping region is written. In this case it is not enough to write the cell target of the write, but also the overlapping cells must be updated with the new value.

Example 2. Continuing Example 1, the assignment $*_{\texttt{unsigned int}}\&V := 5$ updates the value in the cell c_1, but also c_2 must be updated because the region of memory it represents is also affected by the assignment (or its value can be forgotten altogether, which is a simple and sound way to update c_2 and can be efficiently implemented by removing c_2 thanks to the intersection semantics of overlapping cells). □

For this purpose we introduce two operators allowing to add and remove cells to and from the state, while ensuring that the result over-approximates the initial state (according to the byte-level semantics). Notice that it is always sound to create and initialize a new cell with $\top$ as $\top$ does not impose any constraints on the set of values stored in memory, thus it leaves the concretization of the state unchanged. However, because of the overlapping with other cells, it may be sound to initialize the cell with a set of values smaller than $\top$ (crucial to avoid losses of precision). Given a cell c, we denote with $\phi(c, C)$ an expression (where C is the current set of cells) which summarizes the values stored by the other cells in the region identified by c. By construction, $\phi(c, C)$ refers only to other

cells or constant values; in particular it does not contain pointer dereferences. We call this kind of expressions *scalar* expressions. Scalar expressions are generated by the language:

$$\begin{aligned} \text{Scalar-Expr} \ni e ::= \quad & [v_1, v_2] \mid \&V \mid \texttt{NULL} \mid \texttt{invalid} \\ \mid \quad & \langle V, o, \tau \rangle \in \text{Cells} \mid \circ\, e \mid e_1 \diamond e_2 \end{aligned}$$

For any $\langle C, r\rangle$ and c (possibly not in C), ϕ must satisfy the condition

$$\gamma_{Cell}\langle C, r\rangle(c) \subseteq \text{benc}_\tau(E[\![\phi(c, C)]\!]r), \tag{1}$$

where $E[\![\cdot]\!]$ denotes a scalar evaluation, and $\text{benc}_\tau(\cdot)$ is lifted to $\mathcal{P}(\mathbb{I})$. Therefore, it is possible to define the operator add-cell : Cells $\to \mathcal{E}_c \to \mathcal{P}(\mathcal{E}_c)$ as add-cell$(c, \langle C, r\rangle) \triangleq \{\langle C \cup \{c\}, r[c \mapsto v]\rangle \mid v \in E[\![\phi(c, C)]\!]r\}$. It is easy to see that that if ϕ satisfies (1) then the cell addition is exact, i.e., it does not change the concretization of the state. The cell removal operator remove-cell : Cells $\to \mathcal{E}_c \to \mathcal{P}(\mathcal{E}_c)$ can be simply implemented as remove-cell$(c, \langle C, r\rangle) \triangleq \{\langle C \setminus \{c\}, r\big|_{C\setminus\{c\}}\rangle\}$. Notice that due to the intersection semantics, removing a cell corresponds to removing a *constraint*, therefore this can only induce over-approximations. This is sound for the forward semantics presented in this section, but will become unsound for the backward semantics that we present later which instead requires under-approximations.

Because dereferencing a pointer may create new cells, the semantics of expressions must return a cell state in addition to a value, thus it has signature $E_c[\![a]\!] : \mathcal{E}_c \to \mathcal{P}(\mathcal{E}_c \times \mathbb{I})$. The semantics of assignments ensures that no cell overlaps the target cell by removing the cells overlapping the target after it has been updated. This is done with a remove-overlapping-cells operator, which repeatedly calls remove-cell until all the overlapping cells are removed. The cell semantics is reported in Fig. 4.

2.3 Over-approximating Abstract Semantics

Let A be a base numeric abstract domain with concretization $\gamma(\cdot)$, lattice operators $\sqcup^\sharp, \sqcap^\sharp, \nabla^\sharp$, and abstract semantics $\tau^\sharp[\![s]\!]$ defined on a restriction of the language comprising only numeric expressions (i.e., without pointers). In the following, we recall how A can be lifted to a *cell abstract domain* $\widehat{A}$ abstracting the cell semantics. To abstract a concrete invariant $\mathcal{P}(\mathcal{E}_c)$ it is necessary to approximate both the set of cells in the state and the cell's content. The set of cells is abstracted with a single set that is shared among all the concrete cell states. The representation of the set itself does not require abstractions as Cells is finite since $\mathbb{V}$ is assumed to be finite (this limitation will be addressed in Sect. 3). To abstract the content of numeric cells we utilize a numeric dimension in the base domain A, and for pointer cells we abstract the pointer base with a set of possible targets and the offset with a dimension in the numeric base domain. To simplify the notation, we group the numeric and pointer abstractions in a single *scalar* abstraction $\widetilde{A} \triangleq A \times (\text{Cells}_{ptr} \rightharpoonup \mathcal{P}(\mathbb{V} \cup \{\texttt{NULL}, \texttt{invalid}\}))$. Lattice operators are computed pair-wise between A and the pointer map and point-wise

Algorithm 1 Forward abstract transfer functions.

```
1: function FwdAssign(D̂♯, ∗τp := a)
2:     acc ← ⊥̂♯
3:     for all ⟨a_i, l_i⟩ ∈ resolve(a, D̂♯) do
4:         for all ⟨c, l_j⟩ ∈ deref(p, D̂♯) do
5:             ⟨C′, D̃♯′⟩ ← D̂♯
6:             for all ⟨p, ⟨V, o, τ⟩⟩ ∈ l_i :: l_j do
7:                 ⟨C′, D̃♯′⟩ ← add-cell♯(⟨V, o, τ⟩, ⟨C′, D̃♯′⟩)
8:                 ⟨C′, D̃♯′⟩ ← ⟨C′, τ̃♯⟦assume(p = &V + o)⟧D̃♯′⟩
9:             D̃♯′ ← τ̃♯⟦c := a_i⟧D̃♯′
10:            acc ← acc ⊔̂♯ remove-overlapping-cells♯(c, ⟨C′, D̃♯′⟩)
11:    return acc
12: function FwdAssume(D̂♯, assume(a))
13:    acc ← ⊥̂♯
14:    for all ⟨a_i, l_i⟩ ∈ resolve(a, D̂♯) do
15:        ⟨C′, D̃♯′⟩ ← D̂♯
16:        for all ⟨p, ⟨V, o, τ⟩⟩ ∈ l_i do
17:            ⟨C′, D̃♯′⟩ ← add-cell♯(⟨V, o, τ⟩, ⟨C′, D̃♯′⟩)
18:            ⟨C′, D̃♯′⟩ ← ⟨C′, τ̃♯⟦assume(p = &V + o)⟧D̃♯′⟩
19:        D̃♯′ ← τ̃♯⟦assume(a_i)⟧D̃♯′
20:        acc ← acc ⊔̂♯⟨C′, D̃♯′⟩
21:    return acc
```

within pointers of the pointer map, but not necessarily point-wise in A as A may be relational (e.g., polyhedra [19]). In $\widetilde{A}$, numeric statements are handled by A, assignments of pointers are carried out by updating the value in the pointer map of the assigned variable (and the offset in A) and assume statements are carried out by filtering out from the map values that do not satisfy the guard, e.g., $\widetilde{\tau}^\sharp[\![\textbf{assume}(\langle p, 0, \texttt{ptr}\rangle = \texttt{NULL})]\!]$ removes from $\langle p, 0, \texttt{ptr}\rangle$ all non-`NULL` values.

Hence, we have: $\widehat{A} \triangleq \{\langle C, \widetilde{D}^\sharp\rangle \mid C \subseteq \text{Cells}, \widetilde{D}^\sharp \in \widetilde{A}\}$. Notice that all the memory environments abstracted by $\widetilde{D}^\sharp$ are defined over the same set of cells C. The concretization is defined as follows:

$$\widehat{\gamma}^\sharp\langle C, \langle D^\sharp, P\rangle\rangle \triangleq \left\{ \langle C, r\rangle \;\middle|\; \begin{array}{l} \exists \psi \in \gamma(D^\sharp).\ \forall c = \langle V, o, \tau\rangle \in C. \\ \left\{ \begin{array}{ll} r(c) = \psi(c) & \text{if } \tau \neq \texttt{ptr} \\ r(c) = \langle p, \psi(c)\rangle & \text{if } \tau = \texttt{ptr} \wedge p \in P(c) \cap \mathbb{V} \\ r(c) = p & \text{if } \tau = \texttt{ptr} \wedge p \in P(c) \setminus \mathbb{V} \end{array} \right. \end{array} \right\}.$$

To compute lattice operators it is necessary to ensure that both arguments are defined on the same set of cells. This is done by adding the missing cells with an add-cell$^\sharp$: Cells $\to \widehat{A} \to \widehat{A}$ operator which adds and initializes a new cell using the same approach as add-cell. Then lattice operators can be computed on the scalar abstraction (note that Cells is finite, hence there is no convergence problem for the widening operator). The semantics of statements is presented

in the form of an algorithm, see Algorithm 1. Assignments $\widehat{\tau}^\sharp[\![*_\tau p := a]\!]\widehat{D}^\sharp$ and tests $\widehat{\tau}^\sharp[\![\mathbf{assume}(a)]\!]\widehat{D}^\sharp$ are implemented in two steps. Firstly, in an expression we replace (transitively) pointer dereferences with sets of scalar expressions where dereferences are replaced by cells that may be accessed. For instance if $p \mapsto \{\&X, \&Y\}$, then the expression $*_{\texttt{int}}p$ is replaced by $\langle X, 0, \texttt{int}\rangle$ and $\langle Y, 0, \texttt{int}\rangle$ (notice that cells play the roles of variables in scalar expressions). This task is accomplished by an operator resolve : $(\text{Expr} \times \widehat{D}^\sharp) \to \mathcal{P}(\text{Scalar-Expr} \times (\text{Cells}_{ptr} \times \text{Cells})^*)$ which yields pairs made of a scalar expression and sequence of pointer-cell associations that characterize the resolution (e.g., in the example above we have the pointer-cells $\langle\langle p, 0, \texttt{ptr}\rangle, \langle X, 0, \texttt{int}\rangle\rangle$ and $\langle\langle p, 0, \texttt{ptr}\rangle, \langle Y, 0, \texttt{int}\rangle\rangle$). We define also a specialized version of resolve, deref : $(\mathbb{V}_p \times \widehat{D}^\sharp) \to \mathcal{P}(\text{Cells} \times (\text{Cells}_{ptr} \times \text{Cells}))$, which yields a set of cells pointed by a pointer variable (in addition to the pointer-cell association of the derefence). deref is used to resolve the dereference on the left-hand side of assignments: while this could be done with resolve, keeping a dedicated operator will become important in the backward semantics. Secondly, for each pair we execute the statement in the scalar domain and join all the results. Finally, for assignments, overlapping cells are removed with remove-overlapping-cells$^\sharp$ which removes the overlapping cells from the abstract domain.

2.4 Backward Semantics

The semantics proposed in [34] and recalled in the previous section focuses only on forward over-approximating operators. In this section we study a backward semantics based on the same abstraction but computing sufficient pre-conditions.

Byte-level Semantics. In general, the backward version $\overleftarrow{f}$ of a function $f : \mathcal{P}(D) \to \mathcal{P}(D)$ is defined as $\overleftarrow{f}(S) \triangleq \{x \mid f(\{x\}) \subseteq S\}$, that is, it computes a collection of *all and only* the states x that transition to a post-condition $f(\{x\})$ included in S. Notice that this matches the definition of sufficient pre-conditions, previously given for a transition system. Therefore, we can define the backward semantics as the backward version of the forward semantics.

Example 3. Following [35], in a purely numeric setup (i.e., if states are in $\mathbb{V} \to \mathbb{R}$) the definition of backward semantics yields in the case of assignments and tests:

$$\overleftarrow{\tau}[\![v := a]\!]S = \{\psi \mid \forall x \in E[\![a]\!]\psi.\ \psi[v \mapsto x] \in S\}$$
$$\overleftarrow{\tau}[\![\mathbf{assert}(a)]\!]S = S \cup \{\psi \mid E[\![a]\!]\psi \subseteq \{0\}\}$$ □

The *backward semantics* of the byte-level semantics of Fig. 3, denoted $\overleftarrow{\tau_b}[\![\cdot]\!]$, is defined as the backward version of the forward semantics $\tau_b[\![\cdot]\!]$. In particular, the two cases of assignments and filters can be formulated as follows:

$$\begin{aligned}
\overleftarrow{\tau_b}[\![*_\tau p := a]\!]S = \{\rho \mid \forall v \in E_b[\![a]\!]\rho.\ \forall \langle b_0, \ldots, b_{\text{sizeof}(\tau)-1}\rangle \in \text{benc}_\tau(v).\\
\forall \langle V, o\rangle \in E_b[\![p]\!]\rho.\\
\rho[\langle V, o\rangle \mapsto b_0, \ldots, \langle V, o + \text{sizeof}(\tau) - 1\rangle \mapsto b_{\text{sizeof}(\tau)-1}] \in S\}\\
\overleftarrow{\tau_b}[\![\textbf{assume}(a)]\!]S = S \cup \{\rho \mid\ \forall v \in E_b[\![a]\!]\rho.\ (v \in \mathbb{R} \implies v = 0) \wedge\\
(v \in \text{Ptr} \implies v = \texttt{NULL})\}
\end{aligned}$$

Cell-based Semantics. A backward semantics for the cell-based model of memory can be derived as a sound *under-approximation* of the backward byte-level semantics. A simple choice satisfying this condition would be to define the backward semantics as the backward version of the forward semantics. Unfortunately this can be too restrictive if the post-condition contains a set of cells different from the one that naturally arises from the forward semantics.

Example 4. Let $c_1 \triangleq \langle V, 0, \texttt{unsigned short}\rangle$ and $c_2 \triangleq \langle V, 0, \texttt{unsigned char}\rangle$ be two cells and $\sigma \triangleq \langle\{c_1, c_2\}, [c_1 \mapsto 1, c_2 \mapsto 1]\rangle$ be a state. Consider now the assignment s defined as $*_{\texttt{unsigned short}}\&V = *_{\texttt{unsigned short}}\&V + 1$ which, in the forward direction, increments the value of c_1 and removes c_2 (because it overlaps with c_1). If $\overleftarrow{\tau_c}[\![s]\!]$ was defined as the backward version of $\tau_c[\![s]\!]$ then it would contain all and only the states that, after applying $\tau_c[\![s]\!]$, are subsumed by the post-condition, but, as s removes c_2 and the post-condition $\{\sigma\}$ contains only states where c_2 is present, we have $\overleftarrow{\tau_c}[\![s]\!]\{\sigma\} = \varnothing$. On the other hand, the same computation at byte-level yields a non-empty result: the byte-level concretization of σ is the the environment $\rho \triangleq \gamma_{Cell}(\sigma) = [\langle V, 0\rangle \mapsto 1, \langle V, 1\rangle \mapsto 0]$ and $\overleftarrow{\tau_b}[\![s]\!]\{\rho\} = \{[\langle V, 0\rangle \mapsto 0, \langle V, 1\rangle \mapsto 0]\} \neq \varnothing$. □

To avoid this difficulty we give a different definition of backward semantics where the inclusion check of backward functions (i.e., $f(\{x\}) \subseteq S$) is not computed between sets of cell states but between sets of byte-level states obtained by the concretization of the cell states. Formally, for any $S \subseteq \mathcal{E}_c$ and $s \in \text{Stat}$, we have:

$$\overleftarrow{\tau_c}[\![s]\!]S \triangleq \{\sigma \mid \gamma_{Cell}(\tau_c[\![s]\!]\{\sigma\}) \subseteq \gamma_{Cell}(S)\}.$$

Example 5. Continuing Example 4, we analyze s with post-condition $S = \{\sigma\}$, thus $\gamma_{Cell}(S) = \{\rho\}$. The state $\sigma_1 \triangleq \langle\{c_1, c_2\}, [c_1 \mapsto 0, c_2 \mapsto 0]\rangle$ is in the pre-condition as $\gamma_{Cell}(\tau_c[\![s]\!]\{\sigma_1\}) = \gamma_{Cell}(\{\langle\{c_1\}, [c_1 \mapsto 1]\rangle\}) = \{[\langle V, 0\rangle \mapsto 1, \langle V, 1\rangle \mapsto 0]\}$. For the same reason, $\sigma_2 \triangleq \langle\{c_1\}, [c_1 \mapsto 0]\rangle$ and all the other cell states that share the same byte-level concretization of σ_1 are in the pre-condition. □

Proposition 1. *The backward cell-based semantics is sound with respect to the backward byte-level semantics, i.e., for all $s \in$ Stat and $S \in \mathcal{P}(\mathcal{E}_c)$*

$$\gamma_{Cell}(\overleftarrow{\tau_c}[\![s]\!]S) \subseteq \overleftarrow{\tau_b}[\![s]\!](\gamma_{Cell}(S)).$$

Proof. Recall the soundness condition of the forward semantics

$$\forall P \subseteq \mathcal{E}_c.\ \gamma_{Cell}(\tau_c[\![s]\!]P) \supseteq \tau_b[\![s]\!](\gamma_{Cell}(P)) \tag{2}$$

then for all $S \subseteq \mathcal{E}_c$ we have

$$\begin{aligned}
\gamma_{Cell}(\overleftarrow{\tau_c}[\![s]\!]S) &= \gamma_{Cell}(\{\sigma_c \mid \gamma_{Cell}(\tau_c[\![s]\!]\{\sigma_c\}) \subseteq \gamma_{Cell}(S)\}) && \text{[Def. } \overleftarrow{\tau_c}[\![\cdot]\!]\text{]} \\
&\subseteq \gamma_{Cell}(\{\sigma_c \mid \tau_b[\![s]\!]\gamma_{Cell}(\{\sigma_c\}) \subseteq \gamma_{Cell}(S)\}) && \text{[By (2)]} \\
&= \gamma_{Cell}(\{\sigma_c \mid \forall \rho \in \gamma_{Cell}(\{\sigma_c\}).\ \tau_b[\![s]\!]\{\rho\} \subseteq \gamma_{Cell}(S)\}) && \text{[Def. } \tau_b[\![\cdot]\!]\text{]} \\
&= \gamma_{Cell}(\{\sigma_c \mid \forall \rho \in \gamma_{Cell}(\{\sigma_c\}).\ \rho \in \overleftarrow{\tau_b}[\![s]\!]\gamma_{Cell}(S)\}) && \text{[Def. } \overleftarrow{\tau_b}[\![\cdot]\!]\text{]} \\
&= \gamma_{Cell}(\{\sigma_c \mid \gamma_{Cell}(\{\sigma_c\}) \subseteq \overleftarrow{\tau_b}[\![s]\!]\gamma_{Cell}(S)\}) \\
&= \{\gamma_{Cell}(\sigma_c) \mid \gamma_{Cell}(\{\sigma_c\}) \subseteq \overleftarrow{\tau_b}[\![s]\!]\gamma_{Cell}(S)\} && \text{[Def. } \gamma_{Cell}\text{]} \\
&\subseteq \overleftarrow{\tau_b}[\![s]\!]\gamma_{Cell}(S) && \square
\end{aligned}$$

2.5 Lower Abstract Operators

Concrete invariants are of the form $\mathcal{P}(X)$, where X can be the universe of byte-level states or cell states, hence lattice operators such as inclusion check, join and meet coincide with set operators and are all exact. On the contrary, lattice operators of the abstract domain $\widehat{A}$ are in general only sound (in the sense of over-approximation) but not exact, hence they can not be reused in the backward semantics which instead requires operators sound for under-approximations. Consequently, we need to define $\underline{\widehat{\sqcup}}^\sharp : (\widehat{A} \times \widehat{A}) \to \widehat{A}$, called *lower join* and $\underline{\widehat{\sqcap}}^\sharp : (\widehat{A} \times \widehat{A}) \to \widehat{A}$, called *lower meet*, computing an under-approximation of the concrete $\cup$ and $\cap$. We define also a *lower widening* $\underline{\widehat{\nabla}}^\sharp : (\widehat{A} \times \widehat{A}) \to \widehat{A}$ which under-approximates the intersection of its arguments and enforces convergence in a finite number of steps (see [32,35] for examples of lower operators in the case of numeric domains). As a first step we need to ensure that both arguments share the same set of cells. Fortunately the *unification* approach used in $\widehat{\sqcup}^\sharp$, $\widehat{\sqcap}^\sharp$ and $\widehat{\nabla}^\sharp$ (see Paragraph 2.3) is based on the add-cell operator which is exact and thus can be reused here. It remains to study how the scalar abstraction, now defined on the same set of cells, can be joined, met and widened. Unfortunately, a point-wise definition (as we do for over-approximating operators) may be unsound as shown in the following example.

Example 6. Let $c_1 \triangleq \langle X, 0, \texttt{int}\rangle$ and $c_2 \triangleq \langle P, 0, \texttt{ptr}\rangle$ be two cells. Consider $\widehat{D}_1^\sharp \triangleq \langle\{c_1, c_2\}, \langle[c_1 \mapsto [0,5]], [c_2 \mapsto \{\&Y\}]\rangle\rangle$ and $\widehat{D}_2^\sharp \triangleq \langle\{c_1, c_2\}, \langle[c_1 \mapsto [6,10]], [c_2 \mapsto \{\&Z\}]\rangle\rangle$, where for simplicity we omit the offset of P in the numeric abstraction. We have that $[c_1 \mapsto [0,5]] \sqcup^\sharp [c_1 \mapsto [6,10]] = [c_1 \mapsto [0,10]]$ is exact and also $[c_2 \mapsto \{\&Y\}] \sqcup^\sharp [c_2 \mapsto \{\&Z\}] = [c_2 \mapsto \{\&Y, \&Z\}]$ is. Unfortunately, combining the two yields $\langle\{c_1, c_2\}, \langle[c_1 \mapsto [0,10]], [c_2 \mapsto \{\&Y, \&Z\}]\rangle\rangle$ which contains the state $c_1 = 2 \wedge c_2 = \&Z$ that is not present in the concrete union. $\square$

This problem stems from cross terms that appear in the computation of the join, if defined point-wise. For example, computing point-wise the join of the

Fig. 5: The join between the red and blue rectangles on the left is not exact, however it is possible to under-approximate the red rectangle along the x-axis with the intersection between the x components of the two rectangles (case (6a)). The result can be joined exactly yielding the rectangle in green on the right.

Cartesian product of two domains, say, $A = A_1 \times A_2$, yields

$$\begin{aligned}
\gamma(\langle D_1 \sqcup_1^\sharp D_1', D_2 \sqcup_2^\sharp D_2' \rangle) &= \gamma_1(D_1 \sqcup_1^\sharp D_1') \times \gamma_2(D_2 \sqcup_2^\sharp D_2') \\
&\supseteq (\gamma_1(D_1) \cup \gamma_1(D_1')) \times (\gamma_2(D_2) \cup \gamma_2(D_2')) \qquad (3) \\
&= (\gamma_1(D_1) \times \gamma_2(D_2)) \cup (\gamma_1(D_1') \times \gamma_2(D_2')) \cup \\
&\quad (\gamma_1(D_1) \times \gamma_2(D_2')) \cup (\gamma_1(D_1') \times \gamma_2(D_2)) \\
&\supseteq (\gamma_1(D_1) \times \gamma_2(D_2)) \cup (\gamma_1(D_1') \times \gamma_2(D_2')) \qquad (4) \\
&= \gamma(\langle D_1, D_2 \rangle) \cup \gamma(\langle D_1', D_2' \rangle)
\end{aligned}$$

where the undesired cross terms are colored in red. To design an under-approximating join it is necessary to ensure that the cross terms are included in the others terms or equivalently that the inclusion (4) becomes an equality. Conditions for this to occur were studied for example in [2]. In this simple case, it is enough to require that $(D_2 \sqsubseteq_2^\sharp D_2' \vee D_1' \sqsubseteq_1^\sharp D_1) \wedge (D_2' \sqsubseteq_2^\sharp D_2 \vee D_1 \sqsubseteq_1^\sharp D_1')$. Therefore, if this condition is satisfied the cross terms are not present and the lower join can be defined point-wise (notice that the inclusion (3) is reversed for lower joins, and thus the definition is sound). Otherwise, the lower join must fall back to either of its two arguments. In summary, we define

$$\langle D_1, D_2 \rangle \sqcup^\sharp \langle D_1', D_2' \rangle \triangleq \begin{cases}
\langle D_1', D_2' \rangle & \text{if } D_1 \sqsubseteq_1^\sharp D_1' \wedge D_2 \sqsubseteq_2^\sharp D_2' \quad (5a) \\
\langle D_1, D_2 \rangle & \text{if } D_1' \sqsubseteq_1^\sharp D_1 \wedge D_2' \sqsubseteq_2^\sharp D_2 \quad (5b) \\
\langle D_1, D_2 \sqcup_2^\sharp D_2' \rangle & \text{if } D_1 = D_1' \quad (5c) \\
\langle D_1 \sqcup_1^\sharp D_1', D_2 \rangle & \text{if } D_2 = D_2' \quad (5d) \\
\langle D_1, D_2 \rangle \text{ or } \langle D_1', D_2' \rangle & \text{otherwise} \quad (5e)
\end{cases}$$

Moreover, if the equality conditions of (5c) and (5d) are not satisfied, it is possible to under-approximate one component of one of the two arguments until they are equal and thus enforce the equality (see Fig. 5). A simple way to do so is to replace one component with the meet of the two arguments. Consequently, we can replace cases (5c) and (5d) respectively with (6a) and (6b) defined as

follows:

$$\begin{cases} \langle D_1 \sqcap_1^\sharp D_1', D_2 \sqcup_2^\sharp D_2' \rangle & \text{if } \; D_1 \sqcap_1^\sharp D_1' \neq \bot \quad (6a) \\ \langle D_1 \sqcup_1^\sharp D_1', D_2 \sqcap_2^\sharp D_2' \rangle & \text{if } \; D_2 \sqcap_2^\sharp D_2' \neq \bot \quad (6b) \end{cases}$$

The same issue is not present with $\sqcap^\sharp$ and $\underline{\nabla}^\sharp$ which instead can be defined in a point-wise fashion starting from lower operators of A_1 and A_2, e.g., $\langle D_1, D_2 \rangle \sqcap^\sharp \langle D_1', D_2' \rangle \triangleq \langle D_1 \sqcap_1^\sharp D_1', D_2 \sqcap_2^\sharp D_2' \rangle$. This construction can be extended to products of arbitrarily many domains and thus can be used to design lower join, meet and widening for $\widehat{A}$.

Example 7. Continuing Example 6, we can see that all the conditions of $\sqcup^\sharp$ are not satisfied, thus the lower join returns either argument. If we replace $\widehat{D}_2^\sharp$ with $\widehat{D}_2^{\sharp\prime} \triangleq \langle \{c_1, c_2\}, \langle [c_1 \mapsto [6, 10]], [c_2 \mapsto \{\&Y, \&Z\}] \rangle \rangle$ we can notice that $\{\&Y\} \cap \{\&Y, \&Z\} = \{\&Y\} \neq \varnothing$, meaning that (6b) is satisfied. Therefore, we can (lower) join the numeric component and meet the pointer one, which yields the lower join $\langle \{c_1, c_2\}, \langle [c_1 \mapsto [0, 10]], [c_2 \mapsto \{\&Y\}] \rangle \rangle$. □

2.6 Abstract Semantics of Statements.

The forward computation of abstract assignments and tests starts by first *resolving* pointer dereferences and then passing the scalar expressions to the scalar domain. Likewise, in the backward semantics it is necessary to start as well by resolving pointer dereferences. On the other hand, this can not be done using the post-condition since the computation of the assignment or test could have modified the pointers involved in the resolution.

Example 8. Consider the statement $\texttt{arr}[\texttt{arr}[\texttt{0}]] := \texttt{1}$ where `arr` has type `int`$[2]$. If the pre-condition is $\texttt{arr}[\texttt{0}] = \texttt{0}$ then the pointer resolution yields $\{\texttt{arr}[\texttt{0}]\}$ (here we display only the expression component) and thus the post-condition ensures that $\texttt{arr}[\texttt{0}] = \texttt{1}$. However, if the backward analysis used the post-condition to resolve the array index it would erroneously resolve $\texttt{arr}[\texttt{arr}[\texttt{0}]]$ with $\{\texttt{arr}[\texttt{1}]\}$ which is incorrect. □

On the other hand it is always sound to perform the pointer resolution using $\top$ or any over-approximation of the concrete pre-condition. This stems from the fact that using an over-approximation for pointer resolution yields over-approximated expressions and in general in the backward semantics we have that $f \subseteq g$ implies $\overleftarrow{g} \subseteq \overleftarrow{f}$. Consequently, the backward version of an over-approximation of f is an under-approximation of the backward version of f. A sound over-approximation of the pre-condition can be retrieved from a conventional forward analysis, where the analysis invariants are stored for later re-use in the pointer resolution during the backward analysis. With this approach, pointer dereferences are resolved using pointer values stored in the forward over-approximation analysis.

A second issue concerning the analysis of assignments is the need to ensure that cells overlapping the assigned cell are updated or removed.

Example 9. Let $c_1 \triangleq \{V, 0, \texttt{unsigned char}\}$ and $c_2 \triangleq \{V, 0, \texttt{unsigned short}\}$. Consider the state $\widehat{D}^\sharp \triangleq \langle\{c_1, c_2\}, \langle[c_1 \mapsto 1, c_2 \mapsto 1], \_\rangle\rangle$. If the statement $*_{\texttt{unsigned short}}\&V = *_{\texttt{unsigned short}}\&V + 1$ was executed by updating only c_2 we would obtain the pre-condition $\widehat{D}^{\sharp\prime} \triangleq \langle\{c_1, c_2\}, \langle[c_1 \mapsto 1, c_2 \mapsto 0], \_\rangle\rangle$ which is empty because of the intersection semantics. □

This difficulty is sidestepped in the forward semantics by removing the overlapping cells with the remove-overlapping-cells$^\sharp$ operator. Unfortunately, remove-overlapping-cells$^\sharp$ operates by dropping the cells to remove, thus it computes an over-approximation. As simply dropping cells is unsound for under-approximations, we envisage an alternative approach where we firstly transfer the information of the cell we aim to remove to some other cell, after which we can soundly drop the cell, which has become redundant.

Example 10. Consider two overlapping cells $c_1 \triangleq \langle V, 0, \texttt{unsigned char}\rangle$ and $c_2 \triangleq \langle V, 0, \texttt{unsigned short}\rangle$ and a state $\widehat{D}^\sharp \triangleq \langle\{c_1, c_2\}, \langle[c_1 \mapsto [0, 10], c_2 \mapsto [5, 15]], \_\rangle\rangle$. Since c_1 overlaps with the least significant byte of c_2, we have the relation $c_1 = c_2 \,\%\, 256$, hence c_2 can be updated with $(c_2 \,\&\, \texttt{0xff00}) \mid ((c_2 \,\%\, 256) \cap c_1) = [5, 10]$. Then c_1 can be safely removed. □

In general, to determine a relation between cells we can rely on the ϕ function. Consequently, to transfer the information of c_1 to a cell c_2 we can compute the backward semantics of the initialization of c_1 from the values of c_2, that is $\widetilde{\tau}^\sharp[\![c_1 := \phi(c_1, \{c_2\})]\!]$.

Example 11. Consider the state $\widehat{D}^\sharp = \langle C, \widetilde{D}^\sharp\rangle$ of Example 10. To remove c_1, we compute $\widetilde{\tau}^\sharp[\![c_1 := \phi(c_1, \{c_2\})]\!]$, i.e., a set of states such that the initialization of c_1 with an over-approximation of the set of values contained in the memory spanned by c_1 (recall the soundness condition of ϕ, Equation (1)) is contained in $\widetilde{D}^\sharp$, which means that all the information in c_1 is also present in c_2. We have that $\phi(c_1, \{c_2\}) = c_2 \,\%\, 256$ and thus $\widetilde{D}^{\sharp\prime} \triangleq \widetilde{\tau}^\sharp[\![c_1 := c_2 \,\%\, 256]\!][c_1 \mapsto [0, 10], c_2 \mapsto [5, 15]] = [c_1 \mapsto [0, 255], c_2 \mapsto [5, 10]]$. Indeed, in $\widetilde{D}^{\sharp\prime}$ the expression $c_2 \,\%\, 256$ yields $[5, 10]$ which is contained in $\widetilde{D}^\sharp(c_1) = [0, 10]$. □

Proposition 2. *Let $\widehat{D}^\sharp \triangleq \langle C, \widetilde{D}^\sharp\rangle$ be a state and $c \in C$. Define $\widehat{D}^{\sharp\prime}$ as $\langle C \setminus \{c\}, \widetilde{\tau}^\sharp[\![c := \phi(c, C \setminus \{c\})]\!]\widetilde{D}^\sharp\rangle$. Then $\widehat{D}^{\sharp\prime} \mathbin{\widehat{\sqsubseteq}^\sharp} \widehat{D}^\sharp$.*

Proof. In order to check the inclusion $\widehat{D}^{\sharp\prime} \mathbin{\widehat{\sqsubseteq}^\sharp} \widehat{D}^\sharp$, the two arguments must be first unified. In particular, as $\widehat{D}^\sharp$ is defined on the set of cells C, and $\widehat{D}^{\sharp\prime}$ on $C \setminus \{c\}$, the cell c must be added to $\widehat{D}^{\sharp\prime}$, i.e., we obtain $\widehat{D}^{\sharp\prime\prime} \triangleq \text{add-cell}^\sharp(c, \widehat{D}^{\sharp\prime})$. Notice that add-cell$^\sharp$ adds the missing cell in the scalar abstraction by computing $\widetilde{\tau}^\sharp[\![c := \phi(c, C \setminus \{c\})]\!]$. Therefore, by Theorem 2.6 of [35] and soundness of the abstract semantics, $\widetilde{\tau}^\sharp[\![c := \phi(c, C \setminus \{c\})]\!] \circ \widetilde{\tau}^\sharp[\![c := \phi(c, C \setminus \{c\})]\!]\widetilde{D}^\sharp \mathbin{\widetilde{\sqsubseteq}^\sharp} \widetilde{D}^\sharp$, which proves the claim. □

Algorithm 2 Backward abstract transfer functions.

function BwdAssign($\widehat{D}^\sharp, *_\tau p := a, \overline{pre}$)
  $acc \leftarrow \widehat{\top}^\sharp$
  for all $\langle a_i, l_i\rangle \in \text{resolve}(a, \overline{pre})$ **do** ▷ *Resolve rhs with* $\overline{pre}$
    for all $\langle c, l_j\rangle \in \text{deref}(p, \overline{pre})$ **do** ▷ *Deref* p *with* $\overline{pre}$
      $\langle C', \widetilde{D}^{\sharp\prime}\rangle \leftarrow \widehat{D}^\sharp$
      for all $\langle p, \langle V, o, \tau\rangle\rangle \in l_i :: l_j$ **do** ▷ *Add cells found in the resolution*
        $\langle C', \widetilde{D}^{\sharp\prime}\rangle \leftarrow \text{add-cell}^\sharp(\langle V, o, \tau\rangle, \langle C', \widetilde{D}^{\sharp\prime}\rangle)$
      $\widetilde{D}^{\sharp\prime} \leftarrow \widetilde{\overleftarrow{\tau}}^\sharp[\![c := a_i]\!]\widetilde{D}^{\sharp\prime}$ ▷ *Backward assignment*
      for all $\langle p, \langle V, o, \tau\rangle\rangle \in l_i :: l_j$ **do** ▷ *Backward filter of pointers*
        $\widetilde{D}^{\sharp\prime} \leftarrow \widetilde{\overleftarrow{\tau}}^\sharp[\![\textbf{assume}(p = \&V + o)]\!]\widetilde{D}^{\sharp\prime}$
      $acc \leftarrow acc \mathbin{\widehat{\sqcap}^\sharp} \underline{\text{remove-overlapping-cells}^\sharp}(c, \langle C', \widetilde{D}^{\sharp\prime}\rangle)$
  return acc
function BwdAssume($\widehat{D}^\sharp, \textbf{assume}(a), \overline{pre}$)
  $acc \leftarrow \widehat{\top}^\sharp$
  for all $\langle a_i, l_i\rangle \in \text{resolve}(a, \overline{pre})$ **do** ▷ *Resolve* a *with* $\overline{pre}$
    $\langle C', \widetilde{D}^{\sharp\prime}\rangle \leftarrow \widehat{D}^\sharp$
    for all $\langle p, \langle V, o, \tau\rangle\rangle \in l_i$ **do** ▷ *Add cells found in the resolution*
      $\langle C', \widetilde{D}^{\sharp\prime}\rangle \leftarrow \text{add-cell}^\sharp(\langle V, o, \tau\rangle, \langle C', \widetilde{D}^{\sharp\prime}\rangle)$
    $\widetilde{D}^{\sharp\prime} \leftarrow \widetilde{\overleftarrow{\tau}}^\sharp[\![\textbf{assume}(a_i)]\!]\widetilde{D}^{\sharp\prime}$ ▷ *Backward assume*
    for all $\langle p, \langle V, o, \tau\rangle\rangle \in l_i$ **do** ▷ *Backward filter of pointers*
      $\widetilde{D}^{\sharp\prime} \leftarrow \widetilde{\overleftarrow{\tau}}^\sharp[\![\textbf{assume}(p = \&V + o)]\!]\widetilde{D}^{\sharp\prime}$
    $acc \leftarrow acc \mathbin{\widehat{\sqcap}^\sharp} \langle C', \widetilde{D}^{\sharp\prime}\rangle$
  return acc

Based on this result we can design an operator $\underline{\text{remove-cell}^\sharp}$ which removes a cell from a state while ensuring that the result is an under-approximation, and the operator $\underline{\text{remove-overlapping-cells}^\sharp}$ which removes all the cells overlapping a particular cell. This is done by repeatedly calling $\underline{\text{remove-cell}^\sharp}$.

We are now ready to implement the backward semantics of assignments and tests. The semantics is presented in the form of an algorithm, see Algorithm 2. The first step is pointer resolution, which now is performed starting from an over-approximated pre-condition $\overline{pre}$. The rest of the algorithm is similar to the forward version, except for the order (here, first assignments or tests and then filtering of pointers) which is reversed because of the backward direction of the analysis. Finally, the intermediate results are merged with the meet operator. This is because in the computations of sufficient pre-conditions, joins appearing in the forward semantics are replaced by meets. For example $\tau[\![x := x{+}1]\!] \cup \tau[\![x := x - 1]\!]$ maps integers i to $\{i+1, i-1\}$. In order for an integer i to be a sufficient pre-condition for S, both $i + 1 \in S$ and $i - 1 \in S$ must be true, or equivalently $i \in \overleftarrow{\tau}[\![x := x + 1]\!]S \cap \overleftarrow{\tau}[\![x := x - 1]\!]S$.

Proposition 3. *The backward abstract semantics is sound with respect to the backward cell-based semantics, i.e., for all $s \in Stat$ and $\widehat{D}^\sharp \in \widehat{A}$*

$$\widehat{\gamma}^\sharp(\widehat{\overleftarrow{\tau}}^\sharp[\![s]\!]\widehat{D}^\sharp) \subseteq \overleftarrow{\tau_c}[\![s]\!]\widehat{\gamma}^\sharp(\widehat{D}^\sharp).$$

Proof. We outline a sketch of the proof. As in the forward analysis, we need to replace the pointer dereferences in the statement with concrete cells. For this purpose, pointers are resolved with targets in $\overline{pre}$. Indeed, using an over-approximation of the pre-condition will *increase* the number of pointer targets, but, as all intermediate results are eventually met, this will further *restrict* the intersection, thus yielding an under-approximation. After the resolution the algorithm mimics the forward version (in reverse order): we execute the backward analysis of the statement on the scalar abstraction, and then filter the pointers according to the values chosen in the resolution.

3 Heap Abstraction

This section extends the backward semantics studied in Section 2 with support for dynamic memory allocations, which are modeled with the well-known heap recency abstraction [3]. This abstraction is relatively simple yet can handle precisely initialization by hand for low-level languages (e.g., C or assembly) where newly allocated memory is not systematically initialized. For instance, a simple abstraction which summarizes all the allocations with a single summary block would not analyze precisely the program below:

```
1  int *p = malloc(sizeof(int));
2  *p = 0;
```

Indeed, after the allocation the block is uninitialized (i.e., $\top$) and the assignment updates $*p$ with a weak update, thus $*p$ remains $\top$ instead of being initialized with 0. This motivated the authors of [3] to propose the recency abstraction where the most *recent* allocation is singled out in a dedicated block, so that it can be accessed with strong updates, and the older ones are folded in a single summary base. In the program above, the allocation in line 1 yields a recent base, thus the assignment in line 2 can perform a strong update which replaces $\top$ with 0.

Extended Summarization Operators. In the recency abstraction some memory blocks can summarize several concrete memory allocations. To manipulate them, we need some additional statements, namely $\mathbf{expand}(v_1, v_2)$ which copies the constraints of the variable v_1 to a new variable v_2, $\mathbf{fold}(v_1, v_2)$ which removes v_2 and adds its value as a potential value for v_1 and $\mathbf{rename}(v_1, v_2)$ which renames v_1 as v_2. Additionally, it can be shown that $\overleftarrow{\mathbf{expand}(v_1, v_2)} = \mathbf{fold}(v_1, v_2)$ and $\overleftarrow{\mathbf{fold}(v_1, v_2)} = \mathbf{expand}(v_1, v_2)$. See [22] for further details.

Abstraction. The language is extended with atomic statements for memory allocation and de-allocation:

$$\text{Stat} \ni s ::= \ldots \mid v := \text{malloc}() \mid \text{free}(v),$$

where $v \in \mathbb{V}_p$. We call Base the set of memory bases, i.e., memory blocks that can be accessed with pointers (up to this point we considered only variable bases, but this will be extended with heap allocations). Let us denote each allocation site appearing syntactically in the program with a unique identifier, e.g., malloc_{10}. The set of all allocations is $\text{Alloc} \triangleq \{\text{malloc}_1, \ldots, \text{malloc}_n\}$. For each allocation malloc_i we define two bases, namely, b_i^R and b_i^S representing respectively the most recently allocated memory block and the summary block. Consequently, the extended set of bases is $\text{Base} \triangleq \mathbb{V} \cup \{b_i^R \mid \text{malloc}_i \in \text{Alloc}\} \cup \{b_i^S \mid \text{malloc}_i \in \text{Alloc}\}$, and the set of pointers is $\text{Ptr} \triangleq (\text{Base} \times \mathbb{N}) \cup \{\texttt{NULL}, \texttt{invalid}\}$. Notice that if a base is not yet allocated (e.g., before malloc is executed), there is no content at its memory addresses. To account for this, the domain of byte-level states is now a partial map $\mathcal{E}_b : \text{Addr} \rightharpoonup \mathbb{B}$.

The heap recency abstraction does not require the storage of information, and thus it does not possess its own lattice. Conversely, lattice operators are handled by calling lattice operators of the underlying cell abstraction. In particular, this may necessitate unifications if the arguments are defined on different sets of bases. This is achieved by adding the missing bases (and cells) to each argument.

Forward Semantics. The abstract semantics of $v := \text{malloc}_i()$ assigns always to v the recent base, i.e., b_i^R, but it ensures also that such recent base is fresh by (possibly) folding the current content into the summary base. Formally, given a state $\widehat{D}^\sharp$, if $b_i^S \in \widehat{D}^\sharp$, then the state $\widehat{\tau}^\sharp[\![v := b_i^R]\!] \circ \widehat{\tau}^\sharp[\![\mathbf{fold}(b_i^R, b_i^S)]\!]\widehat{D}^\sharp$ is returned, otherwise if $b_i^S \notin \widehat{D}^\sharp$ the state $\widehat{\tau}^\sharp[\![v := b_i^R]\!] \circ \widehat{\tau}^\sharp[\![\mathbf{rename}(b_i^R, b_i^S)]\!]\widehat{D}^\sharp$ is returned. The abstract semantics of $\text{free}(p)$ collects first all the bases of the pointers abstracted by p with offset 0 (so that they point to the base of the memory allocation). For all recent bases, say, b_i^R, free must de-allocate the last allocation, and thus the new recent base should now represent an older allocation: if a summary base is present it will contain all the previous allocations, thus it is expanded into the recent base, otherwise the recent allocation is the only allocation, and thus it can be removed. Formally, given $\widehat{D}^\sharp$, if $b_i^S \in \widehat{D}^\sharp$ then the state $\widehat{\tau}^\sharp[\![\mathbf{expand}(b_i^S, b_i^R)]\!]\widehat{D}^\sharp$ is returned, otherwise if $b_i^S \notin \widehat{D}^\sharp$ then $\widehat{\tau}^\sharp[\![\mathbf{remove}(b_i^R)]\!]$ is returned. If b is a summary base then it means that some allocation was freed, but others may be still present, thus the memory block must be retained (i.e., free is a no-op). Finally, the results for all bases are joined together.[3]

Backward Semantics. As a preliminary observation, notice that the backward semantics of $v := \text{malloc}_i()$ can yield two results, depending on the state of the memory before the allocation.

[3] For the sake of brevity, we omit the handling of runtime errors such as use-after-free or double free, but they are handled in the implementation.

Example 12. Consider the following program.

```
1  int *p = NULL;
2  for (int i = 0; i < 5; i++) {
3      p = malloc(sizeof(int));
4  }
```

Before the malloc is executed in the first iteration of the loop, neither the recent nor the summary base exist in the abstract state. At the start of the second iteration, only the recent base is present, and in subsequent iterations both are present. Note that in the latter two cases the post-condition has the same form, i.e., with both recent and summary bases present. □

Therefore, if both the recent and the summary bases are present in the state, the backward semantics can return a pre-condition which may or may not contain the summary base. Since both pre-conditions are sound, we decide to return the larger one, i.e., the one containing both bases.

We are now ready to present the backward semantics of $v := \text{malloc}_i()$. Given a state $\widehat{D}^\sharp$, three cases are possible:

$$\begin{cases} \widehat{\overleftarrow{\tau}}^\sharp[\![\mathbf{fold}(b_i^R, b_i^S)]\!] \circ \widehat{\overleftarrow{\tau}}^\sharp[\![v := b_i^R]\!]\widehat{D}^\sharp & \text{if } b_i^R \in \widehat{D}^\sharp \wedge b_i^S \in \widehat{D}^\sharp \quad (7a) \\ \widehat{\overleftarrow{\tau}}^\sharp[\![\mathbf{add}(b_i^R)]\!] \circ \widehat{\overleftarrow{\tau}}^\sharp[\![v := b_i^R]\!]\widehat{D}^\sharp & \text{if } b_i^R \in \widehat{D}^\sharp \wedge b_i^S \notin \widehat{D}^\sharp \quad (7b) \\ \bot & \text{if } b_i^R \notin \widehat{D}^\sharp \wedge b_i^S \notin \widehat{D}^\sharp \quad (7c) \end{cases}$$

Notice that the case $b_i^R \notin \widehat{D}^\sharp \wedge b_i^S \in \widehat{D}^\sharp$ is invalid as the summary base is only added if there is already a recent one. The backward semantics of free(p), given a state $\widehat{D}^\sharp$, computes for all bases b_i, such that p abstracts a pointer $\langle b_i, 0\rangle$,

$$\begin{cases} \widehat{\overleftarrow{\tau}}^\sharp[\![\mathbf{expand}(b_i^S, b_i^R)]\!]\widehat{D}^\sharp & \text{if } b_i^R \in \widehat{D}^\sharp \wedge b_i^S \in \widehat{D}^\sharp \quad (8a) \\ \bot & \text{if } b_i^R \in \widehat{D}^\sharp \wedge b_i^S \notin \widehat{D}^\sharp \quad (8b) \\ \widehat{\overleftarrow{\tau}}^\sharp[\![\mathbf{remove}(b_i^R)]\!]\widehat{D}^\sharp & \text{if } b_i^R \notin \widehat{D}^\sharp \wedge b_i^S \notin \widehat{D}^\sharp \quad (8c) \end{cases}$$

and then all the results are met.

Example 13. Consider the program of Fig. 6, which performs memory allocations in line 2. The backward analysis starts with $m = 5$, sufficient pre-condition for the assertion to fail. The free of p falls in case (8a), and thus reverts the expansion of b_1^S into b_1^R. Similarly, the allocations inside AllocAssign (called at line 6) fall in case (7a) and revert the fold of b_1^R into b_1^S. The results of the backward analysis are summarized in Fig. 7. Notice that the content of p is omitted as it always points to b_1^R. □

4 Implementation and Experiments

Our work was implemented in the MOPSA static analysis tool [26]. MOPSA can target several languages, but we focused on the C analysis. It was extended

```
1: function AllocAssign(n)
2:     p ← malloc_1()
3:     *p ← n
4:     return p
5: n ← input(0, 10)
6: p ← AllocAssign(n + 1)
7: m ← *p
8: free(p)
9: assert(m ≠ 5)
```

Fig. 6: Simple program with allocations.

Line	b_1^R	b_i^S	n	m
9	$[0, 10]$	$[0, 10]$	$[0, 10]$	$[5, 5]$
8	$\top$	$[0, 10]$	$[0, 10]$	$[5, 5]$
7	$[5, 5]$	$[0, 10]$	$[0, 10]$	$\top$
6	$[0, 10]$	$[0, 10]$	$[4, 4]$	$\top$

Fig. 7: Backward analysis of the program of Fig. 6.

with the features presented in this work: backward semantics of the cell domain (Sect. 2.4) and heap recency domain (Sect. 3). Additionally, we extended the core of MOPSA to support backward analyses (previously, only the forward direction was supported in its iterators). In particular, we had to adapt the signatures of domains, implement a mechanism for storing and re-calling forward invariants, and finally implement the actual backward iterators and under-approximating domains (intervals, cells and heap). Even if the analyzer supports relational domains, more work is needed to support them in a backward analysis (e.g., improve the scalability). On the contrary we utilize a simple configuration with only intervals domain as it is sufficient to analyze the C language (only floats are not supported).

To improve the precision and efficiency of the backward analysis, a conventional over-approximation forward analysis is executed first, and the computed invariants are stored for re-use in the backward pass. Indeed, some backward abstract transfer functions can benefit from having access to an over-approximation of the pre-condition. For instance, assignments and tests (see Algorithm 2) perform pointer resolution using $\overline{pre}$ which restricts significantly the set of possible targets.

The analysis starts from a $\bot$ post-condition, and when it encounters an operation that may cause a runtime error it computes the states that trigger the error and joins them to the current abstract state. For example, given the post-condition $x \in [0, 10]$, the analysis of the statement **assert**$(x \geq 0)$ will join $x \in [0, 10]$ with $x \in [-\infty, -1]$ (corresponding to a runtime error), thus obtaining $x \in [-\infty, 10]$. This way, we infer a sufficient pre-condition such that, if the control reaches the assertion, it will definitely fail.

Pre-conditions on Inputs. A sufficient pre-condition for a runtime error must ensure that all the program trajectories reaching the error location trigger the error. On the other hand, some bugs may occur only if some external condition is met. For instance, the division by zero error of the program in Fig. 8 is triggered only if `RAND32()` $= 0$. In order to generate counter-examples as well for these cases, we leverage the abstraction of inputs presented in [32], which

```c
void CWE369_Divide_by_Zero__int_rand_divide_01_bad()
{
    int data;
    /* Initialize data */
    data = -1;
    /* POTENTIAL FLAW: Set data to a random value */
    data = RAND32();
    /* POTENTIAL FLAW: Possibly divide by zero */
    printIntLine(100 / data);
}
```

Fig. 8: Task from the Juliet suite where a runtime error depends on some external condition.

allows computing pre-conditions relating to the outcome of calls to an `input`() function. Notice that even if the outcomes of calls to `input`() are collected at the beginning of the program, `input`() can be called anywhere in the program. The input values that we collect are such that, if the program is run and `input`() returns the collected values, the program will either diverge or hit a runtime error.

Benchmarks. We run our analysis on a part of the NIST Juliet collection of benchmarks [5]. This collection consists of 64,099 C/C++ test cases organized under 118 different CWEs. Each task comprises *bad* and *good* functions. Bad functions contain one instance of CWE, while good functions are safe. We analyze 13,261 C tasks, extracted from 12 CWEs that are related to undefined behaviors. In particular, we considered all tasks from those CWEs but the ones where the flaw is not related to runtime errors. Since we are interested in showing bugs, we have analyzed only bad functions. Juliet tasks can use some features of the standard library, e.g., reading from files, memory allocation. We provide some simple stubs for these functions.

Results. For each task, the analysis can present two outcomes: *success* if the analyzer successfully manages to find a non-empty pre-condition; *imprecise* if the analyzer finds an empty pre-condition. A residual part of tasks ("Unsupported" column) could not be analyzed because floats are not supported in the backward analysis. All the analyses were run on an Intel Core i7-1370P CPU with 32 GiB of memory. A timeout of 20 s was set, but all the tasks were analyzed (or failed) before such limit. We report the results in Table 1.

The analyzed tasks mix simple pointer accesses (e.g., read/write from and to aggregate types or raw pointers) and string operations. While the analysis is precise on the former category (thanks to the cell domain and heap recency abstraction), our analysis struggles with string operations, because they can access simultaneously a potentially large amount of cells. This is the case with common string functions, e.g., `strlen`, `strcpy`, that are widely used in the tasks

Code	Title	Time (hh:mm:ss)	Task count	Success	Imprecise	Unsupported
CWE121	Stack-based Buffer Overflow	00:30:37	2508	12%	82%	4%
CWE122	Heap-based Buffer Overflow	00:21:35	1556	17%	80%	2%
CWE124	Buffer Underwrite	00:09:14	758	14%	84%	0%
CWE126	Buffer Over-read	00:08:38	600	23%	74%	2%
CWE127	Buffer Under-read	00:08:44	758	9%	90%	0%
CWE190	Integer Overflow	00:25:10	3420	50%	48%	0%
CWE191	Integer Underflow	00:19:07	2622	57%	42%	0%
CWE369	Divide By Zero	00:04:35	497	56%	35%	8%
CWE415	Double Free	00:01:25	190	63%	36%	0%
CWE416	Use After Free	00:01:60	118	25%	74%	0%
CWE469	Illegal Pointer Subtraction	00:00:14	18	0%	100%	0%
CWE476	NULL Pointer Dereference	00:01:14	216	66%	33%	0%
	Total	02:17:41	13261	35%	62%	2%

Table 1: Experimental results of the analysis of Juliet tasks. For each supported CWE we report the cumulative analysis CPU time of the category, the number of tasks in the category and the outcomes of the analysis.

(e.g., in the CWE469 category the analysis fails due to an imprecision in the analysis of `strchr`). To accurately analyze these tasks it is necessary to track further information about strings, e.g., their length as in [27], and provide precise stubs for these operations (and this is not implemented for the backward analysis yet). This explains the poor results in buffer-related categories as they often involve several complex string manipulations, whereas categories relying less on them (e.g., integer under/overflow) display better results.

5 Related Works

In this section we discuss previous work on backward and under-approximating analyses and memory abstractions.

Backward Analysis and Under-Approximation. As a starting point of our work, we consider the backward analysis of [35,32] that allows to compute sufficient pre-conditions. Traditionally, backward analyses employed in the abstract interpretation field [6,16,17,18] focused on the inference of *necessary* pre-conditions. The two kinds of analyses can yield different results in the presence of non-determinism: sufficient pre-conditions ensure that all the terminating program traces reach the post-condition, whereas necessary pre-conditions only ensure that there exists a trajectory reaching the post-condition.

As noted by the authors of [1], designing under-approximating abstract domains can be a difficult task as they need to be closed under union. Several approaches sidestep this difficulty by adopting higher-order constructions: for instance Lev-Ami et al. [30] consider set-complements of ordinary domains, Schmidt [43] proposes existential quantification and Moy et al. [36] propose a disjunctive completion. In our work we use conventional domains and design a new set of under-approximating operators. Compared to our approach, these

methods retain the best abstraction, but can incur other limitations, namely: less interesting shapes, too high complexity or are difficult to abstract away.

Incorrectness logic [47,37] was recently proposed as an under-approximation version of the well-known Hoare logic [25,20]. Reasoning with it can be made automatic with theorem provers, e.g., in the static analysis tool Pulse [29]. Like us, these works focus on bug-catching rather than verification, but unlike us, they study a forward analysis whereas our analysis works in the backward direction. Therefore, while our analysis can provide pre-conditions for triggering a runtime error, these analyses provide erroneous post-conditions at the error location. Unfortunately, inferring post-conditions for loops is hard in general, and many of these approaches handle loops by unrolling, meaning that bugs can only be discovered if they occur within the maximum number of unrollings. On the contrary our approach relies on widening operators and thus can handle unbounded loops.

Finally, Urban et al. [46,40] study a backward analysis for inference of ranking functions, where the co-domain is under-approximated. The two approaches are not easy to compare as they use different abstractions and semantics.

Memory Abstractions. Several (over-approximating) pointer analyses for C have been proposed in literature: these include flow-insensitive algorithms [44,45,24], i.e., algorithms that do not distinguish different fields in the same struct and flow-sensitive ones such as [28,4,23,48]. They have been applied with success to compiler's design and program analysis as well. Unlike us, these works focus on points-to analysis in isolation, on the contrary our framework combines it with a value analysis (in this case operated in the backward direction).

A large body of work has been devoted to the analysis of the shape of memory allocations: a successful class of works is based on separation logic [41] such as the tools SpaceInvader [9,10] and Infer [8]. Recently, following the interest in incorrectness reasoning [47,37,7], separation logic has been formulated also in an under-approximation fashion (Incorrectness Separation Logic (ISL)) to allow reasoning about the presence of bugs in programs dealing with memory [39,29].

Moreover, separation logic-based methods have been proposed also in the abstract interpretation field [12,11,31,21] enabling for instance the development of the tool MemCAD [42]. In our work we consider a much simpler representation of memory where for each allocation site we smash together all (but the most recent) allocations [3]. Finally, these methods focus on over-approximation forward analyses compared to our approach which focuses on backward under-approximations.

6 Conclusions

In this article we extend the work of [35,32], studying how well-known memory abstractions for the analysis of C, namely the cell domain and the recency domain, can be re-used in a backward under-approximating analysis. We implemented our work in the MOPSA analyzer and assessed its performance by analyzing part of the NIST Juliet collection of benchmarks.

Computing under-approximations by abstract interpretation is still a largely unexplored field, and a wide gap is present with computing over-approximations. This is especially true with respect to the analysis of realistic C programs, possibly involving complex invariants. In this work we demonstrated some preliminary results in this direction, and more sophisticated abstractions are left as future work. Future directions include more advanced abstract domains (e.g., abstraction of memory's shape, string length and content), combination of domains (e.g., partitioning techniques and reduced products), abstraction of the external environment (e.g., resource allocation, more precise abstractions of input streams), and an implementation of backward iterators for the stub language of MOPSA [38].

Acknowledgments. This work was supported by the SECUREVAL project. The SECUREVAL project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-22-PECY-0005.

References

1. Ascari, F., Bruni, R., Gori, R.: Limits and difficulties in the design of under-approximation abstract domains. In: Bouyer, P., Schröder, L. (eds.) Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13242, pp. 21–39. Springer (2022). https://doi.org/10.1007/978-3-030-99253-8_2
2. Bagnara, R., Hill, P.M., Zaffanella, E.: Exact join detection for convex polyhedra and other numerical abstractions. Computational Geometry **43**(5), 453–473 (2010). https://doi.org/10.1016/j.comgeo.2009.09.002
3. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) Static Analysis. pp. 221–239. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11823230_15
4. Balatsouras, G., Smaragdakis, Y.: Structure-sensitive points-to analysis for c and c++. In: Rival, X. (ed.) Static Analysis. pp. 84–104. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_5
5. Black, P.E.: Juliet 1.3 test suite: Changes from 1.2. US Department of Commerce, National Institute of Standards and Technology ... (2018), https://samate.nist.gov/SARD/test-suites/112
6. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. p. 46–55. PLDI '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/155090.155095, https://doi.org/10.1145/155090.155095
7. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A correctness and incorrectness program logic. J. ACM **70**(2) (mar 2023). https://doi.org/10.1145/3582267
8. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of c programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 459–465. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33

9. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Space invading systems code. In: Hanus, M. (ed.) Logic-Based Program Synthesis and Transformation. pp. 1–3. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00515-2_1
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6) (dec 2011). https://doi.org/10.1145/2049697.2049700
11. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 247–260. POPL '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1328438.1328469
12. Chang, B.Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Nielson, H.R., Filé, G. (eds.) Static Analysis. pp. 384–401. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_24
13. Cousot, P.: Principles of Abstract Interpretation. The MIT Press (2021)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
15. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 269–282. POPL '79, Association for Computing Machinery, New York, NY, USA (1979). https://doi.org/10.1145/567752.567778
16. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. The Journal of Logic Programming **13**(2), 103–179 (1992). https://doi.org/https://doi.org/10.1016/0743-1066(92)90030-7
17. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Automated Software Engg. **6**(1), 69–95 (jan 1999). https://doi.org/10.1023/A:1008649901864
18. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 150–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_12
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 84–96. POPL '78, Association for Computing Machinery, New York, NY, USA (1978). https://doi.org/10.1145/512760.512770, https://doi.org/10.1145/512760.512770
20. Floyd, R.W.: Assigning Meanings to Programs, pp. 65–81. Springer Netherlands, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_4
21. Giet, J., Ridoux, F., Rival, X.: A product of shape and sequence abstractions. In: Hermenegildo, M.V., Morales, J.F. (eds.) Static Analysis. pp. 310–342. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-44245-2_15
22. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 512–529. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_38

23. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of c/c++ programs. In: Ranzato, F. (ed.) Static Analysis. pp. 148–168. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_8
24. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. p. 54–61. PASTE '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/379605.379665
25. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (oct 1969). https://doi.org/10.1145/363235.363259
26. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 1–18. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_1
27. Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in c programs. In: Podelski, A. (ed.) Static Analysis. pp. 243–262. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-99725-4_16
28. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 278–289. PLDI '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250766
29. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527325
30. Lev-Ami, T., Sagiv, M., Reps, T., Gulwani, S.: Backward analysis for inferring quantified preconditions (01 2007)
31. Li, H., Rival, X., Chang, B.Y.E.: Shape analysis for unstructured sharing. In: Blazy, S., Jensen, T. (eds.) Static Analysis. pp. 90–108. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_6
32. Milanese, M., Miné, A.: Generation of violation witnesses by under-approximating abstract interpretation. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 50–73. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-50524-9_3
33. Milanese, M., Miné, A.: Artifact of paper: "Under-approximating Memory Abstractions" (May 2024). https://doi.org/10.5281/zenodo.11217437
34. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems. p. 54–63. LCTES '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1134650.1134659
35. Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Science of Computer Programming **93**, 154–182 (2014). https://doi.org/10.1016/j.scico.2013.09.014, special Issue on Invariant Generation
36. Moy, Y.: Sufficient preconditions for modular assertion checking. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 188–202. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_18

37. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371078
38. Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: Pichardie, D., Sighireanu, M. (eds.) Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12389, pp. 223–247. Springer (2020). https://doi.org/10.1007/978-3-030-65474-0_11, https://doi.org/10.1007/978-3-030-65474-0_11
39. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O'Hearn, P., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 225–252. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_14
40. Remil, N.M., Urban, C., Miné, A.: Automatic detection of vulnerable variables for ctl properties of programs. In: Bjorner, N., Heule, M., Voronkov, A. (eds.) Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 100, pp. 116–126. EasyChair (2024). https://doi.org/10.29007/dnpx, https://easychair.org/publications/paper/n5Rq
41. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
42. Rival, X.: https://www.di.ens.fr/~rival/memcad.html
43. Schmidt, D.A.: A calculus of logical relations for over- and underapproximating static analyses. Science of Computer Programming **64**(1), 29–53 (2007). https://doi.org/https://doi.org/10.1016/j.scico.2006.03.008, special issue on the 11th Static Analysis Symposium - SAS 2004
44. Steensgaard, B.: Points-to analysis by type inference of programs with structures and unions. In: Proceedings of the 6th International Conference on Compiler Construction. p. 136–150. CC '96, Springer-Verlag, Berlin, Heidelberg (1996). https://doi.org/10.5555/647473.727458
45. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 32–41. POPL '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/237721.237727
46. Urban, C., Ueltschi, S., Müller, P.: Abstract interpretation of ctl properties. In: Podelski, A. (ed.) Static Analysis. pp. 402–422. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-99725-4_24
47. de Vries, E., Koutavas, V.: Reverse hoare logic. In: Barthe, G., Pardo, A., Schneider, G. (eds.) Software Engineering and Formal Methods. pp. 155–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24690-6_12
48. Wang, W., Barrett, C., Wies, T.: Partitioned memory models for program analysis. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 539–558. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_29