Abstraction of memory block manipulations by symbolic loop folding

Jérôme Boillot, Jérôme Feret

Abstraction of memory block manipulations by symbolic loop folding

Jérôme Boillot and Jérôme Feret

DIENS, École Normale Supérieure, PSL University, CNRS, Inria, Paris, France
jerome.{boillot,feret}@ens.fr

Abstract. We introduce a new abstract domain for analyzing memory block manipulations, focusing on programs with dynamically allocated arrays. This domain computes properties universally quantified over the value of the loop counters, both for assignments and tests. These properties consist of equalities and comparison predicates involving abstract expressions, represented as affine forms in the loop counters and symbolic dereferences. All these methods have been incorporated within the `Astrée`© static analyzer that checks for the absence of run-time errors in embedded critical software. We also give insights on how to implement this abstract domain within any other C static analyzer.

Keywords: Array analysis · Loop folding · Symbolic methods · Abstract interpretation

1 Introduction

As increasingly complex programs, such as operating systems, are successfully formally verified, the memory properties we aim to prove become progressively more subtle. The framework of abstract interpretation has been prolific at providing techniques to prove memory properties. However, the state-of-the-art approaches still lack precision when manipulating pointers, in particular when the memory regions are unbounded. As an example, during the bootstrap of the expanded memory, linking the physical memory to the virtual memory requires proving relations between those addresses, which requires extending the expressivity of symbolic reasoning.

In particular, as in the Gauge domain [19], our abstraction introduces the values of loop counters to express more properties. This improves accuracy while scaling to the analysis of large programs. The analysis of memory block manipulations requires aggregating the effect of the assignments and tests of the previous loop iterations. For this, we go beyond by quantifying universally abstract properties over the precedent values of each loop counters.

We present in this paper a symbolic approach that keeps track of the history of the memory manipulations that are done in the loops.

Related Work. Analyzing the content of arrays has been done by using several methods with different degrees of precision, scalability, and automation. A first

one consists in *smashing* all the cells of the array into a single abstract value [3]. Then, the pointed value can represent more than one element and the updates are weak. To improve the precision, new techniques have been introduced such as dimension summarization [7], and dynamically partitioning arrays [8,10,6] to express relational properties about segments of arrays. More recently, the analysis of matrix manipulations considered in [11] was one of the first considering multidimensional arrays. However, those techniques do not allow expressing relationships among values and offsets.

The idea of the creation of this new abstract domain comes from two ideas. First, from the Gauge domain [19] that computes, for each variable, bounds that are affine forms in the loop counters. Then, by combining such reasoning with symbolic methods [16,4], it becomes possible to express properties about parts of the memory unknown at analysis time.

Shape analysis is broadly used to reason about memory manipulation [13]. However, it heavily relies on inductive type definitions specified in separation logics [18]. Whereas our abstraction provides several grains of precision for describing memory states, shape analysis tends to lose too much information when the inductive types are not satisfied or cannot be proved.

Outline. We sketch as follows the outline of the paper. In Sect. 2, we give the syntax and the semantics of the toy imperative and untyped language that we will use all along this paper. Symbolic manipulations of expressions often require additional syntactic constructors to abstract parts of the expressions and to account for the information computed by the other parts of the analysis. In Sect. 3, we introduce the syntax and the semantics of the abstract expressions that are handled by our abstract domain. In Sect. 4, we define the abstraction of sets of memory states, while in Sect. 5, we describe the abstract transformers which lift the computation of the concrete semantics in the abstract. In Sect. 6, we describe the result of our analysis on three case-studies (namely, a matrix transposition, the bootstrap of the paging in an x86 operating system, and an array look-up procedure). In Sect. 7, we discuss the implementation details and the current limitations of our framework. We conclude in Sect. 8.

2 Syntax & Semantics

2.1 Syntax of a toy imperative and untyped language

In this section we present a toy imperative and untyped language.

As presented in Fig. 1, an expression is either a constant in $\mathbb{Z}$, the address of a variable that belongs to the finite set of variables $\mathcal{V}$, the value located at the address pointed by another expression, or a sum/difference/product of two expressions.

The statements are labelled by program labels $\ell \in \mathcal{L}$ that are distinct and follow a total order $\leq_{\mathcal{L}}$ that corresponds to their order of appearance in the program. Statements can be either a no-op, a sequence of statements, an assignment, or a conditional branching/loop with the condition that corresponds to

the comparison of two expressions. In the case of the assignment, the left-value is the dereference of the address whose content will be modified by the assignment, and the right-value is either an expression or the dynamic allocation of a memory block of size of the expression given as a parameter.

In addition, we introduce the same syntactic sugar as in the C language to ease the understanding of programs: the value of a variable is the value located at its address, and accessing the $e_2{}^{\text{th}}$ element of e_1 corresponds to dereferencing the address $e_1 + e_2$.

$$V \stackrel{\text{def}}{=} *[\&V]$$
$$e_1[e_2] \stackrel{\text{def}}{=} *[e_1 + e_2]$$

prog	::=	$^{\ell}$*stmt* $^{\ell'}$	(*program*)
$^{\ell}$*stmt* $^{\ell'}$	::=	$^{\ell}$**skip**$^{\ell'}$	(*no-op*)
	|	$^{\ell}$*stmt* ; $^{\ell_1}$*stmt* $^{\ell'}$	(*sequence*)
	|	$^{\ell}$*lval* := *rval* $^{\ell'}$	(*assignment*)
	|	$^{\ell}$**if** *expr* $\bowtie$ *expr* **then** $^{\ell_t}$*stmt* $^{\ell'_t}$ **else** $^{\ell_f}$*stmt* $^{\ell'_f}$ **endif**$^{\ell'}$	(*conditional*)
	|	$^{\ell}$**while** *expr* $\bowtie$ *expr* **do** $^{\ell_b}$*stmt* $^{\ell'_b}$ **done**$^{\ell'}$	(*loop*)
expr	::=	$c \mid \&V \mid *[expr] \mid expr \diamond expr$	(*arithmetic expression*)
lval	::=	$*[expr]$	(*left-value*)
rval	::=	*expr* | **malloc**(*expr*)	(*right-value*)
$\bowtie$	::=	$= \mid \neq \mid < \mid \leq \mid > \mid \geq$	(*arithmetic comparison operator*)
$\diamond$	::=	$+ \mid - \mid \times$	(*binary arithmetic operator*)
c	$\in$	$\mathbb{Z}$	(*constant*)
V	$\in$	$\mathcal{V}$	(*variable*)
ℓ	$\in$	$\mathcal{L}$	(*program label*)

Fig. 1. Syntax of an untyped imperative language

```
1 int n;
2 int *arr =
3 malloc(n * sizeof(int));
4 for (int i=0; i<n; i++) {
5   arr[i] = i;
6 }
```

```
ℓ1 *[&arr] := malloc(*[&n]);
ℓ2 *[&i] := 0;
ℓ3 while *[&i] < *[&n] do
ℓ4   *[*[&arr] + *[&i]] := *[&i];
ℓ5   *[&i] := *[&i] + 1
ℓ6 done ℓ7
```

```
ℓ1 arr := malloc(n);
ℓ2 i := 0;
ℓ3 while i < n do
ℓ4   arr[i] := i;
ℓ5   i := i + 1
ℓ6 done ℓ7
```

Fig. 2. Representation of a simple C function's body in the language of the analyzer

To present the language in more details we depict on the left of Fig. 2 a simple C function's body that fills an array of size `n` such that each cell contains

its own index. In the middle of the figure, we wrote an equivalent program in the language that we consider. Note that **malloc**(n) allocates an array of n elements in $\mathbb{Z}$. Because of this idealization, we do not have to multiply n by the number of bytes on which each element is represented. Then, on the right part of the figure we wrote the same program but with the syntactic sugar described previously.

2.2 Concrete semantics of the language

We now describe the *concrete semantics* of our language, that is a mathematical description of its behaviors.

This semantics is non-standard, because we use loop counters as ghost variables (*i.e.*, variables that do not appear in the program) to track how many times each loop body has been evaluated since the most recent evaluation of the loop's entry point. While these counters are not used in the concrete semantics, they play a crucial role in the abstract semantics, where they allow expressing properties universally quantified over loop counter values. In particular, loop counters are useful to express inductive invariants and prove the soundness of the analysis.

A *memory state* is a triple of functions $(\rho, \sigma, \phi) \in \mathcal{R} \stackrel{\text{def}}{=} S_\rho \times S_\sigma \times S_\phi$.

- The first component ρ is a map from addresses in $\mathbb{N}$ to the corresponding value in $\mathbb{Z}$. It corresponds to considering that the memory is an infinite array with naturals as addresses, and integers as values. Then, $S_\rho \stackrel{\text{def}}{=} \mathbb{N} \to \mathbb{Z}$. Let us skip the second element for now.
- The third component ϕ associates each loop counter $\lambda_\ell \in \Lambda$ to its value, that is the number of times the corresponding loop labelled by ℓ has been executed. Then $S_\phi : \Lambda \rightharpoonup \mathbb{N}$. Notice that the function is partial, because before entering a loop and after leaving it the corresponding loop counter does not exist in the domain of ϕ.
- Finally, the second component σ maps each variable, and each dynamically allocated block, to its corresponding address in ρ and its size that represents the number of cells that are reserved to it starting from this address. As dynamically allocated blocks are identified by the program label of the associated **malloc**$(\cdot)$ statement and the value of the loop counters at that time, we get $S_\sigma \stackrel{\text{def}}{=} (\mathcal{V} \cup (\mathcal{L} \times S_\phi)) \rightharpoonup (\mathbb{N} \times \mathbb{N})$.

Definition 1 (Initial set of memory states). *The initial set of memory states is:*

$$R_0 \stackrel{\text{def}}{=} \left\{ (\rho_0, \sigma_0, \phi_0) \in \mathcal{R} \;\middle|\; \begin{array}{l} \mathrm{dom}(\sigma_0) = \mathcal{V},\ \mathrm{dom}(\phi_0) = \varnothing,\ \forall v_1, v_2 \in \mathrm{dom}(\sigma_0), \\ (a_1, s_1) \stackrel{\text{def}}{=} \sigma_0(v_1),\ (a_2, s_2) \stackrel{\text{def}}{=} \sigma_0(v_2), \\ s_1 = 1 \ \wedge\ (v_1 \neq v_2 \implies a_1 \neq a_2) \end{array} \right\}$$

The definition of the initial set of memory states R_0 in Def. 1 that is made of:

- ρ_0 that is any function from $\mathbb{N}$ to $\mathbb{Z}$,

- σ_0 that maps only the variables to their corresponding memory blocks that consist in a non-deterministic address in $\mathbb{N}$ and a constant size of 1, but such that the same address is not given to two different variables,
- ϕ_0 that is the partial function with empty domain. Such functions will be denoted $\varnothing$ in the rest of the article.

$$
\begin{array}{ll}
[\![c]\!](\rho,\sigma) & \stackrel{\text{def}}{=} c \\
[\![\&V]\!](\rho,\sigma) & \stackrel{\text{def}}{=} a \ \textit{with}\ (a,s) \stackrel{\text{def}}{=} \sigma(V) \\
[\![*[e]]\!](\rho,\sigma) & \stackrel{\text{def}}{=} \rho([\![e]\!](\rho,\sigma)) \\
[\![e_1 \diamond e_2]\!](\rho,\sigma) & \stackrel{\text{def}}{=} [\![e_1]\!](\rho,\sigma) \diamond [\![e_2]\!](\rho,\sigma)
\end{array}
$$

$$
\begin{array}{ll}
\{\!|\, {}^{\ell}\mathbf{skip}^{\ell'} \,|\!\}R & \stackrel{\text{def}}{=} R \\
\{\!|\, {}^{\ell}s_1\,;{}^{\ell_1}s_2\,{}^{\ell'} \,|\!\}R & \stackrel{\text{def}}{=} (\{\!|\, {}^{\ell_1}s_2\,{}^{\ell'} \,|\!\} \circ \{\!|\, {}^{\ell}s_1\,{}^{\ell_1} \,|\!\})R \\
\{\!|\, {}^{\ell}\mathbf{if}\ e_1 \bowtie e_2\ \mathbf{then}\ {}^{\ell_\mathtt{t}}s_\mathtt{t}\,{}^{\ell'_\mathtt{t}}\ \mathbf{else}\ {}^{\ell_\mathtt{f}}s_\mathtt{f}\,{}^{\ell'_\mathtt{f}}\ \mathbf{endif}^{\ell'} \,|\!\}R \stackrel{\text{def}}{=} & \\
\qquad (\{\!|\, {}^{\ell_\mathtt{t}}s_\mathtt{t}\,{}^{\ell'_\mathtt{t}} \,|\!\} \circ \mathit{filter}_{\ell,\mathtt{t}}(e_1 \bowtie e_2))R \ \cup\ (\{\!|\, {}^{\ell_\mathtt{f}}s_\mathtt{f}\,{}^{\ell'_\mathtt{f}} \,|\!\} \circ \mathit{filter}_{\ell,\mathtt{f}}(e_1 \not\bowtie e_2))R & \\
\{\!|\, {}^{\ell}\mathbf{while}\ e_1 \bowtie e_2\ \mathbf{do}\ {}^{\ell_b}s\,{}^{\ell'_b}\ \mathbf{done}^{\ell'} \,|\!\}R \stackrel{\text{def}}{=} (\mathit{forget}_\ell \circ \mathit{filter}_{\ell,\mathtt{f}}(e_1 \not\bowtie e_2)) & \\
\qquad \left(\bigcup_{n\in\mathbb{N}} (\mathit{inc}_\ell \circ \{\!|\, {}^{\ell_b}s\,{}^{\ell'_b} \,|\!\} \circ \mathit{filter}_{\ell,\mathtt{t}}(e_1 \bowtie e_2))^n \mathit{new}_\ell(R) \right) &
\end{array}
$$

$$
\{\!|\, {}^{\ell}*[e_1] \coloneqq e_2\,{}^{\ell'} \,|\!\}R \stackrel{\text{def}}{=} \{\, (\rho[a \coloneqq v],\sigma,\phi) \mid (\rho,\sigma,\phi) \in R,\ a \stackrel{\text{def}}{=} [\![e_1]\!](\rho,\sigma),\ v \stackrel{\text{def}}{=} [\![e_2]\!](\rho,\sigma) \,\}
$$

$$
\{\!|\, {}^{\ell}*[e_1] \coloneqq \mathbf{malloc}(e_2)\,{}^{\ell'} \,|\!\}R \stackrel{\text{def}}{=}
\left\{ (\rho[a_l \coloneqq a_r],\sigma[(\ell,\phi) \coloneqq (a_r,s_r)],\phi) \;\middle|\; \begin{array}{l} (\rho,\sigma,\phi) \in R, \\ a_l \in \mathbb{N} : a_l = [\![e_1]\!](\rho,\sigma), \\ s_r \in \mathbb{N} : s_r = [\![e_2]\!](\rho,\sigma), \\ a_r \in \mathbb{N} : \forall (a_o,s_o) \in \mathrm{im}(\sigma), \\ \quad [a_r, a_r{+}s_r) \ \cap\ [a_o, a_o{+}s_o) = \varnothing \end{array} \right\}
$$

$$
\begin{array}{ll}
\mathit{filter}_{\ell,v}(e_1 \bowtie e_2)R & \stackrel{\text{def}}{=} \{\, (\rho,\sigma,\phi) \in R \mid [\![e_1]\!](\rho,\sigma) \bowtie [\![e_2]\!](\rho,\sigma) \,\} \\
\mathit{new}_\ell(R) & \stackrel{\text{def}}{=} \{\, (\rho,\sigma,\phi[\lambda_\ell \coloneqq 0]) \mid (\rho,\sigma,\phi) \in R \,\} \\
\mathit{inc}_\ell(R) & \stackrel{\text{def}}{=} \{\, (\rho,\sigma,\phi[\lambda_\ell \coloneqq \phi(\lambda_\ell)+1]) \mid (\rho,\sigma,\phi) \in R \,\} \\
\mathit{forget}_\ell(R) & \stackrel{\text{def}}{=} \{\, (\rho,\sigma,\phi \restriction_{\mathrm{dom}(\phi)\setminus\{\lambda_\ell\}}) \mid (\rho,\sigma,\phi) \in R \,\}
\end{array}
$$

Fig. 3. Semantics of the language as a denotational semantics

We now introduce the concrete semantics in Fig. 3. We first define the concrete semantics of expressions $[\![\mathit{expr}]\!] : \mathcal{R} \rightharpoonup \mathbb{Z}$. It maps a memory state to the evaluation of the expression in that memory state, that is an element of $\mathbb{Z}$, if the dereferences are legal. The evaluation of a constant just returns the constant, the evaluation of the address of a variable V returns the address that was stored as the first element of $\sigma(V)$, dereferencing an expression corresponds to evaluating the address corresponding to the expression in parameter and retrieving the value at this address in the memory map ρ. Then, evaluating binary arithmetic

operators consists in evaluating both operands and applying the operator on the evaluations.

Then, we define the concrete semantics of statements $\{\!|\, stmt \,|\!\} : \wp(\mathcal{R}) \to \wp(\mathcal{R})$, that maps each set of memory states to another one on which the transformations corresponding to the statements have been performed.

2.3 Consistency of the semantics

The consistency of the semantics relies on the preservation of two properties. The first one, established by Theorem 1, concerns the absence of aliases between the different allocated regions. The second one, established by Theorem 2, ensures that each allocation identifier is introduced at most once, along a given trace.

Firstly, we introduce few auxiliary definitions to specify the scope of loop counters, to state the absence of aliasing in allocated memory regions, and to state that the allocation identifiers are consistent with the current execution.

Definition 2 (Alive loop counters at each program label). *For every $\ell \in \mathcal{L}$, let Λ_ℓ be the subset of Λ that represents all the loop counters alive at the program point ℓ,*

$$\Lambda_\ell \stackrel{\text{def}}{=} \{\, \lambda_{\ell_1} \in \Lambda \mid \ell_1 <_{\mathcal{L}} \ell <_{\mathcal{L}} \overline{\ell_1} \,\}$$

with $\forall \lambda_{\ell_1} \in \Lambda, \overline{\ell_1}$ the program label just after the end of the loop statement labelled by ℓ_1. We also define a total order $\leq_\Lambda$ on loop counters that mirrors the order of their corresponding program points, i.e., *$\lambda_\ell \leq_\Lambda \lambda_{\ell'} \Leftrightarrow \ell \leq_{\mathcal{L}} \ell'$.*

Definition 3 (Non-aliasing memory state property). *A set of memory states $R \in \wp(\mathcal{R})$ is said to be* non-aliasing(R) *when none of its allocated memory regions (described by σ in each memory set in R) aliases with another one.*

$$\begin{aligned}&\text{non-aliasing}(R) \stackrel{\text{def}}{=} \forall (\rho, \sigma, \phi) \in R,\ \forall k_1, k_2 \in \text{dom}(\sigma),\\ &\quad (k_1 \in \mathcal{V} \implies s_1 = 1) \ \wedge\ (k_1 \neq k_2 \implies [a_1, a_1{+}s_1) \ \cap\ [a_2, a_2{+}s_2) = \varnothing)\\ &\text{with } (a_1, s_1) \stackrel{\text{def}}{=} \sigma(k_1), \text{ and } (a_2, s_2) \stackrel{\text{def}}{=} \sigma(k_2).\end{aligned}$$

In particular, non-aliasing(R_0).

Definition 4 (Program label and loop counters identify memory allocations).

$$\begin{aligned}&\text{alloc-consistent}(\ell, R) \stackrel{\text{def}}{=} \forall (\rho, \sigma, \phi) \in R, \forall \ell_1 \in \mathcal{L}, \forall \phi' : \text{dom}(\phi) \to \mathbb{N},\\ &\quad (\ell \leq_{\mathcal{L}} \ell_1 \ \wedge\ \phi \preccurlyeq \phi') \implies (\ell_1, \phi') \notin \text{dom}(\sigma)\end{aligned}$$

with $\preccurlyeq$ the lexicographic order of the loop counter values scanned in the order described by $\leq_\Lambda$ (defined in Def. 2). That is to say, two functions ϕ and ϕ' are considered as equal if they map the same value for each loop counter, otherwise they are ordered as the values of the minimal loop counter (with respect to the total order $\leq_\Lambda$) that maps to different values. Intuitively, $\phi \preccurlyeq \phi'$ when ϕ' is reachable only after ϕ.

In particular, $\forall \ell \in \mathcal{L}, \text{alloc-consistent}(\ell, R_0)$.

This also implies that, before executing a statement ${}^{\ell}e_1 := \textbf{malloc}(e_2)\,{}^{\ell'}$, $(\ell, \phi) \notin \text{dom}(\sigma)$: the identifier of the newly allocated memory block is used for the first time.

Theorem 1 (Concrete semantics does not introduce aliasing). *Let $R \in \wp(\mathcal{R})$ be a set of memory states and ${}^{\ell}s\,{}^{\ell'}$ be a statement, then*

$$\text{non-aliasing}(R) \implies \text{non-aliasing}(\{\!|\, {}^{\ell}s\,{}^{\ell'} \,|\!\}R)$$

This directly implies that, if p is a program, then $\text{non-aliasing}(\{\!|\, p \,|\!\}R_0)$.

Theorem 2 (Consistency of the identification of memory allocations). *Let $R \in \wp(\mathcal{R})$ be a set of memory states and ${}^{\ell}s\,{}^{\ell'}$ be a statement, then*

$$\text{alloc-consistent}(\ell, R) \implies \text{alloc-consistent}(\ell', \{\!|\, {}^{\ell}s\,{}^{\ell'} \,|\!\}R)$$

This directly implies that if $p \stackrel{\text{def}}{=} {}^{\ell}s\,{}^{\ell'}$ is a program then $\text{alloc-consistent}(\ell, \{\!|\, p \,|\!\}R_0)$.

Theorems 1 and 2 state that the properties non-aliasing and alloc-consistent are preserved by the concrete semantics of statement, which means that, if the property holds initially, it will continue to hold at each program label.

3 Abstract expressions

In the following section, we introduce abstract expressions. They stand both for left and right values, and they allow expressing and propagating intermediary reasoning about them.

3.1 Syntax of abstract expressions

We present their grammar in Fig. 4.

The goal of the abstract domain is to output properties that are universally quantified over the loop counter values. Then, we introduce $\Pi \stackrel{\text{def}}{=} \{\, \pi_\ell \mid \lambda_\ell \in \Lambda \,\}$, with $\pi_\ell \in \Pi$ that corresponds to a universally quantified value of λ_ℓ. Typically, we will infer properties like $\forall \pi_\ell \in [0, \lambda_\ell), p(\pi_\ell)$, with p a predicate over the memory.

We introduce the elements of $base^\sharp$ that are either:

- `NULL` which is used to represent expressions that do not depend on the address of a memory block, with `NULL` $+ 0$ that refers to the value 0. It is used to represent integers,
- $\&V$ which corresponds to the address where the value of the variable V is stored,
- $malloc_\ell(\phi)$ which corresponds to the first address of the memory block dynamically allocated by the statement at label ℓ when the loop counters had the values ϕ. This identifies uniquely the memory block because each statement can be evaluated only once for fixed loop counters values (as a consequence of Theorem. 2).

$$
\begin{array}{lcl}
\mathit{rval}^\sharp & ::= & (\mathit{base}^\sharp + \mathit{offset}^\sharp) \\
\mathit{offset}^\sharp & ::= & \mathit{affine}_{\mathit{deref}}[\mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]] \\
\mathit{lval}^\sharp & ::= & *[\mathit{base}^\sharp + \mathit{affine}_{\mathit{ctr}}[\mathit{affine}_{\mathit{deref}}[\mathbb{Z}]]] \\
\mathit{base}^\sharp & ::= & \texttt{NULL} \mid \&V \mid \mathit{malloc}_\ell(\mathit{malloc}_{\mathit{id}}) \\
\mathit{malloc}_{\mathit{id}} & ::= & \bot \mid \lambda_\ell \mapsto \mathit{offset}^\sharp,\ \mathit{malloc}_{\mathit{id}} \\
\mathit{affine}_{\mathit{deref}}[K] & ::= & K \mid K \times *[\mathit{base}^\sharp + \mathit{affine}_{\mathit{deref}}[K]] + \mathit{affine}_{\mathit{deref}}[K] \\
\mathit{affine}_{\mathit{ctr}}[K] & ::= & K \mid K \times \tau_\ell + \mathit{affine}_{\mathit{ctr}}[K] \\
V & \in & \mathcal{V} \\
\lambda_\ell & \in & \Lambda \\
\tau_\ell & \in & \Lambda \cup \Pi
\end{array}
$$

Fig. 4. Syntax of abstract expressions

As the goal is to represent expressions as affine forms of loop counters and dereferences, we introduce $\mathit{affine}_{\mathit{ctr}}[K]$ and $\mathit{affine}_{\mathit{deref}}[K]$ that are parametric non-terminals, with K a non-terminal that can be interpreted as an integer.

$\mathit{affine}_{\mathit{ctr}}[K]$ encodes an affine expression with coefficients in K and variables that are the number of times the corresponding loop has been evaluated. As an example, if $^{\ell}$**while** ... appears in a program, then $(2 \times \lambda_\ell + 1) \in \mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]$ represents the integer value: "twice the number of times the loop's body has been evaluated until now, plus one".

$\mathit{affine}_{\mathit{deref}}[K]$ encodes an affine expression with coefficients in K and variables that are dereferences in the memory block starting from a $\mathit{base}^\sharp$ and an offset that itself is in $\mathit{affine}_{\mathit{deref}}[K]$. As an example, the expression `2 * arr[i+1]` can be represented by $2 \times *[\&\texttt{arr} + 1 \times *[\&\texttt{i} + 0] + 1] + 0$ that is in $\mathit{affine}_{\mathit{deref}}[\mathbb{Z}]$.

From those bricks, we construct the key elements of $\mathit{rval}^\sharp$ and $\mathit{lval}^\sharp$ that are $\mathit{affine}_{\mathit{deref}}[\mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]]$ and $\mathit{affine}_{\mathit{ctr}}[\mathit{affine}_{\mathit{deref}}[\mathbb{Z}]]$. They differ in that only the first one allows loop counters (elements of Λ or Π depending on whether they are generalized) to appear in the dereferences. This difference will be discussed in Sect. 5.

In particular, elements of $\mathit{offset}^\sharp$ represent integers/addresses that are affine forms in symbolic dereferences, with coefficients that are affine forms in loop counters (generalized or not), with coefficients that are in $\mathbb{Z}$. Then, an abstract right-value in $\mathit{rval}^\sharp$ is just an abstract base plus an abstract offset starting from this base. In Example. 1 we show an element of $\mathit{rval}^\sharp$.

Example 1. The translation of the right-value `n-`λ_ℓ`+2`, with `n` a variable and λ_ℓ the value of a loop counter, can be:

$$
\underbrace{(\underbrace{\texttt{NULL}}_{\mathit{base}^\sharp} + \underbrace{\underbrace{1}_{\mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]} \times *[\underbrace{\&n}_{\mathit{base}^\sharp} + \underbrace{0}_{\mathit{affine}_{\mathit{deref}}[\mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]]}] + \underbrace{(-1) \times \lambda_\ell + 2)}_{\mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]}}_{\mathit{affine}_{\mathit{deref}}[\mathit{affine}_{\mathit{ctr}}[\mathbb{Z}]]}}_{\mathit{rval}^\sharp}
$$

We now introduce the evaluation of abstract expressions into $\mathbb{Z}$, provided the concrete memory state. This is important to prove the soundness of the translation into abstract expressions in Theorem. 3. The definitions in Fig. 5 really follow the intuition, except for $S_\varphi \stackrel{\text{def}}{\equiv} (\Lambda \cup \Pi) \rightharpoonup \mathbb{N}$ that is a valuation of the (possibly generalized) loop counters. It will be used in the following section while defining the concretization of universally quantified properties: they have to hold for every φ that is a possible valuation of the generalized loop counters.

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{rval^\sharp} : rval^\sharp \to S_\rho \times S_\sigma \times S_\varphi \rightharpoonup \mathbb{Z} \\
[\![b + off]\!]^\sharp_{rval^\sharp}(\rho,\sigma,\varphi) \stackrel{\text{def}}{=} [\![b]\!]^\sharp_{base^\sharp}(\sigma) + [\![off]\!]^\sharp_{offset^\sharp}(\rho,\sigma,\varphi)
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{offset^\sharp} : offset^\sharp \to S_\rho \times S_\sigma \times S_\varphi \rightharpoonup \mathbb{Z} \\
[\![a]\!]^\sharp_{offset^\sharp}(\rho,\sigma,\varphi) \stackrel{\text{def}}{=} [\![a]\!]^\sharp_{affine_{deref}[affine_{ctr}[\mathbb{Z}]]}([\![\cdot]\!]^\sharp_{affine_{ctr}[\mathbb{Z}]}([\![\cdot]\!]^\sharp_{\mathbb{Z}},\varphi),\rho,\sigma)
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{lval^\sharp} : lval^\sharp \to S_\rho \times S_\sigma \times S_\varphi \rightharpoonup \mathbb{Z} \\
[\![*[b + a]]\!]^\sharp_{lval^\sharp}(\rho,\sigma,\varphi) \stackrel{\text{def}}{=} [\![b]\!]^\sharp_{base^\sharp}(\sigma) + [\![a]\!]^\sharp_{affine_{ctr}[affine_{deref}[\mathbb{Z}]]}([\![\cdot]\!]^\sharp_{affine_{deref}[\mathbb{Z}]}([\![\cdot]\!]^\sharp_{\mathbb{Z}},\rho,\sigma),\varphi)
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{base^\sharp} : base^\sharp \to S_\sigma \rightharpoonup \mathbb{N} \\
[\![\texttt{NULL}]\!]^\sharp_{base^\sharp}(\sigma) \stackrel{\text{def}}{=} 0 \\
[\![\&V]\!]^\sharp_{base^\sharp}(\sigma) \stackrel{\text{def}}{=} a \text{ with } (a,s) \stackrel{\text{def}}{=} \sigma(V) \\
[\![malloc_\ell(\phi)]\!]^\sharp_{base^\sharp}(\sigma) \stackrel{\text{def}}{=} a \text{ with } (a,s) \stackrel{\text{def}}{=} \sigma(\ell, [\![\phi]\!]^\sharp_{malloc_{id}})
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{malloc_{id}} : malloc_{id} \to S_\phi \\
[\![\bot]\!]^\sharp_{malloc_{id}} \stackrel{\text{def}}{=} \varnothing \\
[\![\lambda_\ell \mapsto off, m]\!]^\sharp_{malloc_{id}} \stackrel{\text{def}}{=} \phi \text{ with } \phi(\lambda_{\ell_1}) \stackrel{\text{def}}{\equiv} \begin{cases} [\![off]\!]^\sharp_{offset^\sharp} & \text{if } \ell = \ell_1, \\ [\![m]\!]^\sharp_{malloc_{id}}(\lambda_{\ell_1}) & \text{otherwise.} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{affine_{ctr}[K]} : affine_{ctr}[K] \to (K \rightharpoonup \mathbb{Z}) \times S_\varphi \rightharpoonup \mathbb{Z} \\
[\![k]\!]^\sharp_{affine_{ctr}[K]}([\![\cdot]\!]^\sharp_K,\varphi) \stackrel{\text{def}}{=} [\![k]\!]^\sharp_K \\
[\![k \times \tau_\ell + a]\!]^\sharp_{affine_{ctr}[K]}([\![\cdot]\!]^\sharp_K,\varphi) \stackrel{\text{def}}{=} [\![k]\!]^\sharp_K \times \varphi(\tau_\ell) + [\![a]\!]^\sharp_{affine_{ctr}[K]}([\![\cdot]\!]^\sharp_K,\varphi)
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{affine_{deref}[K]} : affine_{deref}[K] \to (K \rightharpoonup \mathbb{Z}) \times S_\rho \times S_\sigma \rightharpoonup \mathbb{Z} \\
[\![k]\!]^\sharp_{affine_{deref}[K]}([\![\cdot]\!]^\sharp_K,\rho,\sigma) \stackrel{\text{def}}{=} [\![k]\!]^\sharp_K \\
[\![k \times *[b + a_1] + a_2]\!]^\sharp_{affine_{deref}[K]}([\![\cdot]\!]^\sharp_K,\rho,\sigma) \stackrel{\text{def}}{\equiv} \\
\qquad [\![k]\!]^\sharp_K \times \rho\Big([\![b]\!]^\sharp_{base^\sharp}(\sigma) + [\![a_1]\!]^\sharp_{affine_{deref}[K]}([\![\cdot]\!]^\sharp_K,\rho,\sigma) \Big) + [\![a_2]\!]^\sharp_{affine_{deref}[K]}([\![\cdot]\!]^\sharp_K,\rho,\sigma)
\end{array}
$$

$$
\begin{array}{l}
[\![\cdot]\!]^\sharp_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \\
[\![k]\!]^\sharp_{\mathbb{Z}} \stackrel{\text{def}}{=} k
\end{array}
$$

Fig. 5. Evaluation of abstract expressions as integers

3.2 Translation from expressions to abstract right-values

We now define the translation of expressions into abstract right-values. At this point, the translation is purely syntactic and does not depend on the memory state. We denote it $(\!|\cdot|\!) : \mathit{expr} \rightharpoonup \mathit{rval}^\sharp$, and we provide a simple definition in Fig. 6. To that extent, we use the helper functions $\mathit{add}_{\mathit{offset}^\sharp}$ (resp. $\mathit{sub}_{\mathit{offset}^\sharp}$) which adds (resp. returns the difference between) two abstract offsets. Those functions only accumulate the dereferences that appear in the operands and merge the remaining constants. In the case of a difference, we take the opposite of the coefficients in head of the dereferences of the second operand.

$$
\begin{array}{lcl}
(\!| c |\!) & \stackrel{\text{def}}{=} & (\texttt{NULL} + c) \\
(\!| \&V |\!) & \stackrel{\text{def}}{=} & (\&V + 0) \\
(\!| *[e] |\!) & \stackrel{\text{def}}{=} & \begin{cases} (\texttt{NULL} + *[b + \mathit{off}] + 0) \text{ if } (b + \mathit{off}) \stackrel{\text{def}}{=} (\!| e |\!), \\ \mathit{undefined} \text{ otherwise.} \end{cases} \\
(\!| e_1 + e_2 |\!) & \stackrel{\text{def}}{=} & \begin{cases} (b + \mathit{add}_{\mathit{offset}^\sharp}(\mathit{off}_1, \mathit{off}_2)) \text{ if } (b + \mathit{off}_1) \stackrel{\text{def}}{=} (\!| e_1 |\!), (\texttt{NULL} + \mathit{off}_2) \stackrel{\text{def}}{=} (\!| e_2 |\!), \\ (b + \mathit{add}_{\mathit{offset}^\sharp}(\mathit{off}_1, \mathit{off}_2)) \text{ if } (\texttt{NULL} + \mathit{off}_1) \stackrel{\text{def}}{=} (\!| e_1 |\!), (b + \mathit{off}_2) \stackrel{\text{def}}{=} (\!| e_2 |\!), \\ \mathit{undefined} \text{ otherwise.} \end{cases} \\
(\!| e_1 - e_2 |\!) & \stackrel{\text{def}}{=} & \begin{cases} (b + \mathit{sub}_{\mathit{offset}^\sharp}(\mathit{off}_1, \mathit{off}_2)) \text{ if } (b + \mathit{off}_1) \stackrel{\text{def}}{=} (\!| e_1 |\!), (\texttt{NULL} + \mathit{off}_2) \stackrel{\text{def}}{=} (\!| e_2 |\!), \\ (\texttt{NULL} + \mathit{sub}_{\mathit{offset}^\sharp}(\mathit{off}_1, \mathit{off}_2)) \text{ if } (b + \mathit{off}_1) \stackrel{\text{def}}{=} (\!| e_1 |\!), (b + \mathit{off}_2) \stackrel{\text{def}}{=} (\!| e_2 |\!), \\ \mathit{undefined} \text{ otherwise.} \end{cases} \\
(\!| e_1 \times e_2 |\!) & \stackrel{\text{def}}{=} & \mathit{undefined}
\end{array}
$$

Fig. 6. Simple translation from expressions to abstract right-values

The soundness of the translations is expressed by Theorem 3.

Theorem 3 (Soundness of expression translation). *For every expression* $e \in \mathit{expr}$ *and every memory state* $(\rho, \sigma, \phi) \in \mathcal{R}$*,*

$$(b + \mathit{off}) \stackrel{\text{def}}{=} (\!| e |\!) \implies [\![e]\!](\rho, \sigma) = [\![b + \mathit{off}]\!]^\sharp_{\mathit{rval}^\sharp}(\rho, \sigma, \phi)$$

Note that, neither this definition of the translation of dereferences is precise, because the abstract base cannot be different from NULL, nor the translation of multiplications is precise. Indeed, such translations are purely syntactic. They could gain precision by relying on semantic properties provided by abstract domains.

We provide some hints about how to improve the translation of multiplications of elements of $\mathit{offset}^\sharp$. There are two cases:

- if one operand is a constant in $\mathbb{Z}$, then we distribute the multiplication by the constant to the coefficients of the other operand (for example, 5 multiplied by $5 \times \lambda_\ell + 0$ would give $25 \times \lambda_\ell + 0$),

– otherwise, if one operand is an affine form in the loop counters while the other one has no loop counters that appear in head, then we can distribute the multiplication by the affine form to the coefficients of the other operand (for example, $5 \times \lambda_\ell + 0$ multiplied by $1 \times *[\&\mathtt{arr} + 1 \times \lambda_\ell + 0] + 0$ would give $(5 \times \lambda_\ell + 0) \times *[\&\mathtt{arr} + 1 \times \lambda_\ell + 0] + 0$).

4 Description of the abstract memory states

An abstract memory state is represented by an element of $\mathcal{R}^\sharp$ that is a tuple of 5 elements that describe constraints over the memory state. The first component is the program point in which the memory state is observed. The second one expresses properties about the loop counters values. The third and fourth components express properties about the assignments and conditions that were previously evaluated but still effective. The last component describes information about the size of dynamically allocated memory blocks in case they were allocated. In the following section, we provide a detailed explanation of how these constraints are stored. Throughout this depiction, we describe the components of a possible abstract memory state of the program presented in Fig. 8, focusing on the program state at label ℓ_9. This program is given both with C syntax (on the left), and in the language presented in this article (on the right). It makes use of dynamic memory allocation, loops and conditional assignments to cover most of the properties described by the abstract memory state.

To ensure the coherence of abstract elements, the loop counters that appear in an abstract state are constrained by some side conditions about the order of appearance in the program (the order $\leq_\Lambda$). These side-conditions are provided in the explanations of each component of the tuple $R^\sharp \in \mathcal{R}^\sharp$.

$$
\begin{aligned}
\mathcal{R}^\sharp &\stackrel{\text{def}}{\equiv} \mathcal{L} \times \mathit{sections}^\sharp \times \mathit{assigns}^\sharp \times \mathit{guards}^\sharp \times \mathit{mallocs}^\sharp \\
\mathit{sections}^\sharp &\stackrel{\text{def}}{\equiv} \Lambda \rightharpoonup (\mathit{offset}^\sharp \times (\mathit{offset}^\sharp \cup \{+\infty\})) \\
\mathit{assigns}^\sharp &\stackrel{\text{def}}{\equiv} \mathcal{L} \rightharpoonup (\wp(\mathit{gen\text{-}sections}^\sharp) \times (\mathit{branch\text{-}key} \rightharpoonup \mathit{comp}^\sharp) \times \mathit{lval}^\sharp \times \mathit{rval}^\sharp) \\
\mathit{guards}^\sharp &\stackrel{\text{def}}{\equiv} \mathit{branch\text{-}key} \rightharpoonup (\wp(\mathit{gen\text{-}sections}^\sharp) \times \mathit{comp}^\sharp) \\
\mathit{mallocs}^\sharp &\stackrel{\text{def}}{\equiv} \mathcal{L} \rightharpoonup (\mathit{offset}^\sharp \cup \{\top\}) \\
\mathit{gen\text{-}sections}^\sharp &\stackrel{\text{def}}{\equiv} \Pi \rightharpoonup (\mathit{offset}^\sharp \cup (\mathit{offset}^\sharp \times \mathit{offset}^\sharp)) \\
\mathit{branch\text{-}key} &\stackrel{\text{def}}{\equiv} \mathcal{L} \times \{\mathtt{t}, \mathtt{f}\} \\
\mathit{comp}^\sharp &\stackrel{\text{def}}{\equiv} \mathit{rval}^\sharp \times \{=, \neq, <, \leq, >, \geq\} \times \mathit{rval}^\sharp
\end{aligned}
$$

Fig. 7. Definition of the abstract memory states

```
1 int n, m;
2 int *arr = malloc(n * sizeof(int));
3 int i = 0;
4 while (i < n) {
5   if (i != m) {
6       arr[i] = i;
7   }
8   i++;
9 }
```

```
ℓ1  arr := malloc(n);
ℓ2  i := 0;
ℓ3  while i < n do
ℓ4    if i ≠ m then
ℓ5      arr[i] := i
ℓ6    else
ℓ7      skip
ℓ8    endif;
ℓ9    i := i + 1
ℓ10 done ℓ11
```

Fig. 8. Program with a dynamic allocation, a loop, and a conditional assignment

4.1 Abstraction of loop counter sections

The second component of an abstract memory state $s^\sharp : \mathit{sections}^\sharp$ represents an over-approximation of the loop counters values. It is a partial map that associates each loop counter that is alive with its possible values. We recall that a loop counter is alive if it occurs in a loop whose body is being evaluated, that is to say, if ℓ stands for the program label, that this loop counter belongs to Λ_ℓ. The possible values are over-approximated by intervals with $\mathit{offset}^\sharp$ as bounds, with possibly no upper-bound. Additionally, the abstraction of sections is constrained by the following side-condition: the image of a loop counter shall only depend on the values of the loop counters that are defined before (that is to say: for every $\lambda_{\ell'} \in \Lambda$ occurring in the image of λ_ℓ, $\lambda_{\ell'} <_\Lambda \lambda_\ell$). Only the functions satisfying this side-condition will be considered in the paper.

For example, in the program presented in Fig. 8, at label ℓ_9, the abstract memory state's sections could be $\{\, \lambda_{\ell_3} \mapsto (0, 1 \times *[\&\mathtt{n} + 0] + 0) \,\}$. This means that λ_{ℓ_3} has its value within 0 and `n` (excluded).

We now introduce the concretization function of the corresponding part of the abstract memory state. We denote it $\gamma_{\mathit{sections}^\sharp} : \mathit{sections}^\sharp \to \wp(\mathcal{R})$, and it is defined for each element $\phi^\sharp \in \mathit{sections}^\sharp$ as:

$$\gamma_{\mathit{sections}^\sharp}(\phi^\sharp) \stackrel{\text{def}}{=} \left\{ (\rho, \sigma, \phi) \in \mathcal{R} \;\middle|\; \begin{array}{l} \mathrm{dom}(\phi) = \mathrm{dom}(\phi^\sharp),\ \forall \lambda_\ell \in \mathrm{dom}(\phi^\sharp), \\ \begin{cases} \phi(\lambda_\ell) \in \mathbb{N} \cap [o_1, o_2) & \text{if } (\mathit{off}_1, \mathit{off}_2) \stackrel{\text{def}}{=} \phi^\sharp(\lambda_\ell), \\ \phi(\lambda_\ell) \in \mathbb{N} \cap [o_1, +\infty) & \text{if } (\mathit{off}_1, +\infty) \stackrel{\text{def}}{=} \phi^\sharp(\lambda_\ell), \end{cases} \\ \text{with } \forall i, o_i \stackrel{\text{def}}{=} [\![\, \mathit{off}_i \,]\!]^\sharp_{\mathit{offset}^\sharp}(\rho, \sigma, \phi \restriction_{\Lambda_{<\lambda_\ell}}) \end{array} \right\}$$

We notice that the side condition of $\phi^\sharp$ is necessary to ensure the well-formedness of the definition, when evaluating the abstraction of offset expressions.

4.2 Abstraction of assignments and tests

We now define the abstraction of constraints coming from previous assignments and tests. This abstraction relies on the description of regions for the potential

value of generalized loop counters (described by elements of *gen-sections*$^\sharp$), and a tracking of the syntactic comparisons that had to hold to reach a program state (described using *branch-key* and *comp*$^\sharp$ elements).

Elements of gen-sections$^\sharp$. They are used to express properties that apply to multiple values of loop counters. The image of $\pi_\ell \in \Pi$ represents the values π_ℓ can take, with π_ℓ the generalized version of the loop counter value $\lambda_\ell \in \Lambda$. It can be either:

- an abstract offset $off_1 \in offset^\sharp$: it is used to express that a property holds "$\forall \pi_\ell \in \mathbb{N} \cap [off_1, \lambda_\ell)$", with as a side condition off_1 which depends only on the live loop counters (not-generalized) defined before λ_ℓ (that are in $\{\, \lambda_{\ell'} \in \Lambda \mid \lambda_{\ell'} \leq_\Lambda \lambda_\ell \,\}$),
- a couple of abstract offsets $off_1, off_2 \in offset^\sharp$: it is used to express that a property holds "$\forall \pi_\ell \in \mathbb{N} \cap [off_1, off_2)$", with as a side condition off_1 and off_2 which depend only on the live loop counters (not-generalized) defined strictly before λ_ℓ (that are in $\{\, \lambda_{\ell'} \in \Lambda \mid \lambda_{\ell'} <_\Lambda \lambda_\ell \,\}$).

If a live loop counter λ_ℓ does not appear in the domain of the element of *gen-sections*$^\sharp$, it means that the loop counter is not generalized, thus $\pi_\ell = \lambda_\ell$.

We now introduce a helper function that, given the set of live loop counters, an element of *gen-sections*$^\sharp$ and a memory state, returns all the valuation functions of the generalized loop counters. We denote it $\gamma_{gen\text{-}sections^\sharp} : \wp(\Lambda) \times \wp(gen\text{-}sections^\sharp) \times S_\rho \times S_\sigma \times S_\phi \to \wp(\Pi \rightharpoonup \mathbb{N})$, and it is defined for each set of live loop counters $\Lambda_{\ell_0} \subseteq \Lambda$, each element $\Phi_g^\sharp \in \wp(gen\text{-}sections^\sharp)$ and each memory state $(\rho, \sigma, \phi) \in (S_\rho \times S_\sigma \times S_\phi)$ as:

$$\gamma_{gen\text{-}sections^\sharp}(\Lambda_{\ell_0}, \Phi_g^\sharp, \rho, \sigma, \phi) \stackrel{\text{def}}{=}$$

$$\left\{ \varphi : \Pi_{\ell_0} \to \mathbb{N} \;\middle|\; \begin{array}{l} \exists \phi_g^\sharp \in \Phi_g^\sharp,\ \forall \lambda_\ell \in \Lambda_{\ell_0}, \\ \varphi(\pi_\ell) \in \left\{ \begin{array}{ll} \{\, \phi(\lambda_\ell) \,\} & \text{if } \pi_\ell \notin \mathrm{dom}(\phi_g^\sharp), \\ \mathbb{N} \cap [l, \phi(\lambda_\ell)) & \text{if } off_1 \stackrel{\text{def}}{=} \phi_g^\sharp(\pi_\ell) \\ \quad \text{with } l \stackrel{\text{def}}{=} [\![\, off_1 \,]\!]^\sharp_{offset^\sharp}(\rho, \sigma, \phi \restriction_{\Lambda_{\leq \lambda_\ell}}), & \\ \mathbb{N} \cap [o_1, o_2) & \text{if } (off_1, off_2) \stackrel{\text{def}}{=} \phi_g^\sharp(\pi_\ell) \\ \quad \text{with } o_i \stackrel{\text{def}}{=} [\![\, off_i \,]\!]^\sharp_{offset^\sharp}(\rho, \sigma, \phi \restriction_{\Lambda_{<\lambda_\ell}}). & \end{array} \right. \end{array} \right\}$$

Elements of branch-key and comp$^\sharp$. They are used together to represent the validity of comparisons. First, *branch-key* is used to determine where the comparison comes from, that is from which conditional branching (through its program label in $\mathcal{L}$) and with which outcome (`t` if the condition was fulfilled, and `f` otherwise). Then, the comparison is represented by the comparison operator and the two compared abstract right-values.

Elements of assigns$^\sharp$. They describe for each program point ℓ corresponding to an assignment statement that, for each possible value of the generalized loop counters (described by elements of *gen-sections*$^\sharp$), if the proof obligations (described

by an element of $\mathit{branch\text{-}key} \rightharpoonup \mathit{comp}^\sharp$) hold, then the addresses represented by a given abstract left-value point to the values represented by a given abstract right-value. As side conditions, the only loop counters allowed to appear in the abstract left/right-values are the generalized versions of the live loop counters, that are the elements of Π_ℓ.

For example, in the program presented in Fig. 8, at label ℓ_9, the abstract memory state's assignments could be the following:

$$
\begin{array}{l}
\{\ \ell_1 \mapsto (\ \{\,\varnothing\,\}, \varnothing, *[\&\texttt{arr} + 0], (\mathit{malloc}_{\ell_1}(\bot) + 0)),\\
\quad \ell_2 \mapsto (\ \{\,\varnothing\,\}, \varnothing, *[\&\texttt{i} + 0], (\texttt{NULL} + 0)),\\
\quad \ell_9 \mapsto (\ \{\,\{\,\pi_{\ell_3} \mapsto 0\,\}\,\}, \varnothing, *[\&\texttt{i} + 0], (\texttt{NULL} + 1 \times \pi_{\ell_3} + 0)),\\
\quad \ell_5 \mapsto (\ \{\,\varnothing,\ \{\,\pi_{\ell_3} \mapsto 0\,\}\,\},\\
\qquad\qquad \{\,(\ell_4, \texttt{t}) \mapsto (\texttt{NULL} + 1 \times \pi_{\ell_3} + 0,\ \neq,\ \texttt{NULL} + 1 \times *[\&\texttt{m} + 0] + 0)\,\},\\
\qquad\qquad *[\mathit{malloc}_{\ell_1}(\bot) + 1 \times \pi_{\ell_3} + 0],\\
\qquad\qquad (\texttt{NULL} + 1 \times \pi_{\ell_3} + 0))\,\}
\end{array}
$$

Let us describe in details this element of $\mathit{assigns}^\sharp$:

- The first line describes the assignment at program label ℓ_1. At this point, no loop counters are defined, so none of them can be generalized. This is what the first occurrence of the symbol $\varnothing$ indicates. Moreover, this assignment is evaluated unconditionally. This is what is embedded in the second occurrence of the symbol $\varnothing$. The two other elements of the tuple are respectively the left and right values after their translations into abstract forms. The left-value corresponds to the variable `arr` and the right-value to the memory block allocated in the statement labelled by ℓ_1.
- The second line is really similar to the first, except that the left-value is `i` and the right-value is the integer 0.
- The third line differs in the presence of a non-$\varnothing$ $\mathit{gen\text{-}sections}^\sharp$ element. It indicates that the corresponding property holds for all π_{ℓ_3} between 0 and λ_{ℓ_3} (excluded, because the considered abstract memory state is in ℓ_9). Indeed, `i` is equal to the number of times the loop's body has been entirely executed.
- The rest is a bit more complex because the assignment is conditional, and because it might have already been executed for the current value of the loop counter λ_{ℓ_3}. It states that, both for $\pi_{\ell_3} = \lambda_{\ell_3}$ (indicated by the symbol $\varnothing$), but also for all π_{ℓ_3} within 0 and λ_{ℓ_3} (indicated by $\{\,\pi_{\ell_3} \mapsto 0\,\}$), if the condition evaluated at program label ℓ_4 was true (that is $\texttt{i} \neq \texttt{m}$), then the ${\pi_{\ell_3}}^{\text{th}}$ cell of the memory block allocated at program label ℓ_1 points to the value π_{ℓ_3}.

We now introduce the concretization function of the corresponding part of the abstract memory state. We denote it $\gamma_{\mathit{assigns}^\sharp} : \mathit{assigns}^\sharp \to \wp(\mathcal{R})$, and it is

defined for each element $a^\sharp \in \mathit{assigns}^\sharp$ as

$$\gamma_{\mathit{assigns}^\sharp}(a^\sharp) \stackrel{\text{def}}{=} \left\{ (\rho,\sigma,\phi) \in \mathcal{R} \;\middle|\; \begin{array}{l} \forall \ell \in \mathrm{dom}(a^\sharp),\ (\Phi_g^\sharp, c^\sharp, l^\sharp, r^\sharp) \stackrel{\text{def}}{=} a^\sharp(\ell), \\ \forall \varphi \in \gamma_{\mathit{gen\text{-}sections}^\sharp}(\Lambda_\ell, \Phi_g^\sharp, \rho, \sigma, \phi), \\ (\rho,\sigma,\varphi) \vdash c^\sharp \implies \rho([\![l^\sharp]\!]^\sharp_{\mathit{val}^\sharp}(\rho,\sigma,\varphi)) = [\![r^\sharp]\!]^\sharp_{\mathit{rval}^\sharp}(\rho,\sigma,\varphi) \end{array} \right\}$$

with $(\rho,\sigma,\varphi) \vdash c^\sharp$ that represents that all the proof obligations embodied by $c^\sharp$ are satisfied in the memory state (ρ,σ,φ). More formally:

$$(\rho,\sigma,\varphi) \vdash c^\sharp \iff \forall (e_1^\sharp, \bowtie, e_2^\sharp) \in \mathrm{im}(c^\sharp), [\![e_1^\sharp]\!]^\sharp_{\mathit{rval}^\sharp}(\rho,\sigma,\varphi) \bowtie [\![e_2^\sharp]\!]^\sharp_{\mathit{rval}^\sharp}(\rho,\sigma,\varphi)$$

Elements of guards$^\sharp$. They are used to indicate that a conditional branching statement produced the same result across multiple iterations when evaluated within loops. It is represented by, for each comparison and output (represented by an element of *branch-key*), and for each range of values the loop counters can take (described by an element of *gen-sections*$^\sharp$), the comparison that holds (described by an element of *comp*$^\sharp$).

For example, in the program presented in Fig. 8, at label ℓ_9, the abstract memory state's guards could be the following.

$$\{ (\ell_3, \mathtt{t}) \mapsto (\{ \varnothing,\ \{ \pi_{\ell_3} \mapsto 0 \} \}, \\ (\mathtt{NULL} + 1 \times \pi_{\ell_3} + 0,\ <,\ 1 \times *[\&\mathtt{n} + 0] + 0)) \}$$

The domain of this function is the singleton $(\ell_3, \mathtt{t})$ which indicates that the condition labelled by ℓ_3 (that is $\mathtt{i} < \mathtt{n}$) holds for the current value of the loop counter (denoted by the element $\varnothing$), but also for every π_{ℓ_3} between 0 and the current value of the loop counter (denoted by the element $\{ \pi_{\ell_3} \mapsto 0 \}$).

We now introduce the concretization function of the corresponding part of the abstract memory state. We denote it $\gamma_{\mathit{guards}^\sharp} : \mathit{guards}^\sharp \to \wp(\mathcal{R})$, and it is defined for each element $g^\sharp \in \mathit{guards}^\sharp$ as

$$\gamma_{\mathit{guards}^\sharp}(g^\sharp) \stackrel{\text{def}}{=} \left\{ (\rho,\sigma,\phi) \in \mathcal{R} \;\middle|\; \begin{array}{l} \forall (\Phi_g^\sharp, (e_1^\sharp, \bowtie, e_2^\sharp)) \in \mathrm{im}(g^\sharp), \\ \forall \varphi \in \gamma_{\mathit{gen\text{-}sections}^\sharp}(\Phi_g^\sharp, \rho, \sigma, \phi), \\ [\![e_1^\sharp]\!]^\sharp_{\mathit{rval}^\sharp}(\rho,\sigma,\varphi) \bowtie [\![e_2^\sharp]\!]^\sharp_{\mathit{rval}^\sharp}(\rho,\sigma,\varphi) \end{array} \right\}$$

4.3 Abstraction of memory blocks dynamically allocated

The elements of *mallocs*$^\sharp$ describe, for each program point ℓ that is associated to a **malloc** statement, that if for a given valuation of the loop counters at that time the memory allocation was performed, then the size of the corresponding memory block is described by a given *offset*$^\sharp$. If the size cannot be expressed precisely, then $\top$ is used. As side conditions, the only loop counters allowed to appear in the representation are the ones of Λ_ℓ. This is because, as explained in the concrete semantics, the memory allocation is identified uniquely by the program label and the value of the loop counters at that time.

For example, in the program presented in Fig. 8, at label ℓ_9, the corresponding abstract memory state's element could be $\{\, \ell_1 \mapsto (1 \times *[\&\mathtt{n} + 0] + 0) \,\}$. This represents that, if it has been allocated, the memory block allocated at program label ℓ_1 and identified by the loop counters' values at that time, is of size $\mathtt{n}$.

We now introduce the concretization function of the corresponding part of the abstract memory state. We denote it $\gamma_{mallocs^\sharp} : mallocs^\sharp \to \wp(\mathcal{R})$, and it is defined for each element $m^\sharp \in mallocs^\sharp$ as

$$\gamma_{mallocs^\sharp}(m^\sharp) \stackrel{\text{def}}{=}$$
$$\left\{ (\rho, \sigma, \phi) \in \mathcal{R} \;\middle|\; \begin{array}{l} \forall \ell' \notin \mathrm{dom}(m^\sharp),\ \forall \varphi' : \Lambda_{\ell'} \to \mathbb{N},\ (\ell', \varphi') \notin \mathrm{dom}(\sigma) \wedge \\ \forall \ell \in \mathrm{dom}(m^\sharp),\ s^\sharp \stackrel{\text{def}}{=} m^\sharp(\ell),\ s^\sharp \neq \top,\ \forall \varphi : \Lambda_\ell \to \mathbb{N}, \\ (\ell, \varphi) \in \mathrm{dom}(\sigma) \implies s = [\![\, s^\sharp \,]\!]^\sharp_{offset^\sharp}(\rho, \sigma, \varphi) \\ \quad \text{with } (a, s) \stackrel{\text{def}}{=} \sigma(\ell, \varphi) \end{array} \right\}$$

4.4 Combining all the constraints described by abstract memory states

We finally introduce the concretization function of abstract memory states. It is denoted $\gamma : \mathcal{R}^\sharp \to \wp(\mathcal{R})$, and it is defined for each abstract memory state $(\ell, \phi^\sharp, a^\sharp, g^\sharp, m^\sharp) \in \wp(\mathcal{R}^\sharp)$ as

$$\gamma(\ell, \phi^\sharp, a^\sharp, g^\sharp, m^\sharp) \stackrel{\text{def}}{=} \gamma_{sections^\sharp}(\phi^\sharp) \cap \gamma_{assigns^\sharp}(a^\sharp) \cap \gamma_{guards^\sharp}(g^\sharp) \cap$$
$$\gamma_{mallocs^\sharp}(m^\sharp) \cap \{\, R \in \mathcal{R} \mid \text{non-aliasing}(\{\, R \,\}) \wedge \text{alloc-consistent}(\ell, R) \,\}$$

The initial abstract memory state is $R_0^\sharp \stackrel{\text{def}}{=} (\ell_0, \varnothing, \varnothing, \varnothing, \varnothing)$ with ℓ_0 the first program label. It is a sound abstraction of R_0, that is $R_0 \subseteq \gamma(R_0^\sharp)$, because the only property enforced by $\gamma(R_0^\sharp)$ is to be a non-aliasing memory state, which is verified for R_0.

5 Operations on abstract memory states

In this section we describe the main abstract transfer functions that compute a sound approximation of their concrete counterparts. The abstract semantics is described by $\{\!|\, stmt \,|\!\}^\sharp : \mathcal{R}^\sharp \to \mathcal{R}^\sharp$ that is the abstract counterpart of the concrete semantics introduced in Fig. 3. The soundness of the abstract transfer functions is expressed by Theorem 4 and elements of proof are given along their definitions.

Theorem 4 (Soundness of the abstract transfer functions). *For every statement ${}^{\ell}s^{\ell'} \in stmt$ and every abstract memory state $R^\sharp \in \mathcal{R}^\sharp$,*

$$\{\!|\, {}^{\ell}s^{\ell'} \,|\!\} \circ \gamma(R^\sharp) \subseteq \gamma \circ \{\!|\, {}^{\ell}s^{\ell'} \,|\!\}^\sharp(R^\sharp)$$

Definition 5. *We introduce merge $: (\mathcal{R}^\sharp \times \wp(gen\text{-}sections^\sharp) \times rval^\sharp \times \mathcal{R}^\sharp \times \wp(gen\text{-}sections^\sharp) \times rval^\sharp) \rightharpoonup rval^\sharp$ that, given two abstract memory states, sets of universally quantified loop counters, and abstract expressions in $rval^\sharp$, returns*

an abstract expression that matches the semantics of the two original ones when evaluated on their respective domains. We provide an overview of the process that is guided by heuristics:

- *we identify parts of the expressions that trivially match in order to reduce the size of expressions to be merged,*
- *for each loop counter that both maps to a constant value and that is not generalized in only one abstract memory state, we replace the occurrences of that loop counter by the constant,*
- *then, if both expressions are evaluated in the context of abstract memory states in which a loop counter is not generalized and maps to two different constant values, we linearly interpolate the remaining parts of the expression. As an example, if we try to merge the expressions* 0 *(with* $\lambda_\ell = 0$*) and* $\boldsymbol{i}$ *(with* $\lambda_\ell = 1$*), then we output the expression* $\lambda_\ell \times \boldsymbol{i}$ *that is indeed semantically equal to both expressions on their respective abstract memory states.*

Abstract join We describe the abstract transfer function corresponding to the union after a branching at the condition at program point ℓ_0. Given two abstract memory states $(\ell_1, \phi_1^\sharp, a_1^\sharp, g_1^\sharp, m_1^\sharp)$ and $(\ell_2, \phi_2^\sharp, a_2^\sharp, g_2^\sharp, m_2^\sharp)$, it is defined as the following:

- the new section of each loop counter λ_ℓ, if $\forall i, (l_i, u_i) \stackrel{\text{def}}{=} \phi_i^\sharp(\lambda_\ell)$, is (l, u) defined as follows. First the lower-bound l is the minimum between l_1 and l_2 if it exists, otherwise 0 (that is a lower-bound over the values of the loop-counters). Symmetrically, we define u as the maximum value between u_1 and u_2 if it exists, or $+\infty$,
- the universally quantified predicates expressed by the elements of $\mathit{assigns}^\sharp$ and $\mathit{guards}^\sharp$ are joined. For all the elements of the domain that appear in the domain of both operands, the expressions involved (either the left/right-values, or the compared right-values) are merged using the *merge* operator described above. In case there are proof obligations in the property, the compared expressions are also merged. In the case where the assignment/test only appears in one operand, if the test located at label ℓ_0 still exists in the corresponding element of $\mathit{guards}^\sharp$, it means we can keep the property if we add the proof obligation corresponding to the element of $\mathit{guards}^\sharp$,
- lastly, the new element of $\mathit{mallocs}^\sharp$ is just the result of the *merge* operator with no loop counters generalized, or $\top$ if they could not get merged.

Widening Applying consecutive abstract joins to compute the abstract transfer function corresponding to loops could loop forever. To mitigate this problem, after computing the abstract join, if the lower-bound (resp. upper-bound) of a loop counter's section keeps decreasing (resp. increasing), we widen it to 0 (resp. $+\infty$). However, to limit the loss of precision, we use *widening thresholds*. To that matter, during the first loop iterations we record the array lookups, and in particular the abstract expressions the loop counters would have to take so the

offset exceeds the bounds of the memory block. Then, we first try to widen the loop counter bounds to those values. In order to ensure that the suite of abstract memory states is ultimately stationary, we limit the number of such widenings to a fixed number.

Abstract assignments When performing an assignment in the abstract, we first translate the left and right values, then we remove (or replace by its previous value if it can be expressed) all the occurrences of the possibly assigned cell both in the abstract memory state and in the translated left and right values.

To avoid losing the properties about the previous assignments at the same program point but from past iterations, it is necessary to prove that the assigned cell does not alias with the previously assigned cells. To that matter, allowing loop counters to appear only in head of $lval^\sharp$ is helpful: if we can order the loop counters such that each coefficient is bigger than the part of the expression with the smaller loop counters, then the wanted property is proven.

Once the possibly affected cells' contents forgotten/rewritten, we can add the assignment (with no generalized counters) to the element of $assigns^\sharp$ and (if existing) merge the previous and new abstract left/right values while keeping the previous proof obligations. Finally, if the assignment also corresponds to a memory allocation, the size of the memory block has to be merged with the previously existing one. Note that, for the soundness proof, we use that the memory states in the concretization have to be alloc-consistent. This gives us that the memory block being allocated has never been allocated before, so the expression can indeed be merged.

Abstract filter When filtering the abstract memory state by a comparison, we apply the same strategy as in the assignment case: we translate the expressions, and merge them with the existing abstract comparison (if it was already existing) with no loop counters generalized (that is, adding the element $\{\varnothing\}$ to $\Phi_g^\sharp$).

Loop counter handling To define the abstract transfer function for the loop statements, we still need to define the abstract counterparts of new_ℓ, $forget_\ell$, inc_ℓ. The first one only adds the loop counter λ_ℓ to the living counters, with default value 0. The second one tries to determine an abstract expression for the value of λ_ℓ. If it can find one, it is used to replace the loop counter everywhere it was defined, otherwise all the properties that involve it are forgotten. Finally, the section corresponding to the loop counter is removed from the abstract memory state. The third one is a bit more complicated: when the loop counter λ_ℓ is incremented, all its occurrences in the abstract memory state have to be replaced by $\lambda_\ell - 1$. Then, the predicates quantified by π_ℓ that ranges over an interval with λ_ℓ as the upper-bound remain only if the predicate still held for the current value of λ_ℓ. The predicates that weren't quantified by π_ℓ that ranges over an interval with λ_ℓ as the upper-bound but that hold for the current value of λ_ℓ that is a constant expression are generalized with π_ℓ that ranges over the interval with

the current expression of λ_ℓ as the lower bound, and λ_ℓ as the upper-bound. If the current value of λ_ℓ couldn't be proven a constant expression, the predicate is generalized with π_ℓ that ranges over the interval with $\lambda_\ell - 1$ as the lower-bound and λ_ℓ as the upper-bound.

6 Case studies and analysis

In the following section, we describe the properties inferred by the abstract domain presented in the paper on some simple, but still realistic, programs. We first demonstrate the capability of the abstract domain to prove relational properties about multidimensional arrays (here matrices). In a second time, we consider a program that is used to set up the expanded memory of an x86 operating system. In particular, this setting takes into account the memory needed by the BIOS, the bootloader, and the code of the operating system. We are particularly interested in this example that was the initial motivation for this abstract domain. Then, we present a program that computes the index of the first element greater than a given value in a sorted array.

6.1 Relational properties about multidimensional arrays

```
1 int n, i, j;
2 int **mat1 = malloc(n * sizeof(int*));
3 int **mat2 = malloc(n * sizeof(int*));
4 i = 0;
5 while (i < n) {
6   mat1[i] = malloc(n * sizeof(int));
7   mat2[i] = malloc(n * sizeof(int));
8   i++;
9 }
10 i = 0;
11 while (i < n) {
12   j = 0;
13   while (j < n) {
14       mat2[i][j] = mat1[j][i];
15       j++;
16   }
17   i++;
18 }
```

```
ℓ1  mat1 := malloc(n);
ℓ2  mat2 := malloc(n);
ℓ3  i := 0;
ℓ4  while i < n do
ℓ5    mat1[i] := malloc(n);
ℓ6    mat2[i] := malloc(n);
ℓ7    i := i + 1
ℓ8  done;
ℓ9  i := 0;
ℓ10 while i < n do
ℓ11   j := 0;
ℓ12   while j < n do
ℓ13     mat2[i][j] := mat1[j][i];
ℓ14     j := j + 1
ℓ15   done;
ℓ16   i := i + 1
ℓ17 done ℓ18
```

Fig. 9. Transposition of a square matrix

The program depicted in Fig. 9 computes the transposition a square matrix `mat1` into another one `mat2`. The abstract domain presented in this paper is able to prove that, at the end of the program, the following properties about the

assignments hold:

$$\begin{cases} \texttt{mat1} = \mathit{malloc}_{\ell_1}(\bot) \wedge \\ \texttt{mat2} = \mathit{malloc}_{\ell_2}(\bot) \wedge \\ \forall \pi_{\ell_4} \in [0,n), \underbrace{*[\mathit{malloc}_{\ell_1}(\bot) + \pi_{\ell_4}]}_{\texttt{mat1}[\pi_{\ell_4}]} = \mathit{malloc}_{\ell_5}(\lambda_{\ell_4} \mapsto \pi_{\ell_4}, \bot) \wedge \\ \forall \pi_{\ell_4} \in [0,n), \underbrace{*[\mathit{malloc}_{\ell_2}(\bot) + \pi_{\ell_4}]}_{\texttt{mat2}[\pi_{\ell_4}]} = \mathit{malloc}_{\ell_6}(\lambda_{\ell_4} \mapsto \pi_{\ell_4}, \bot) \wedge \\ \forall \pi_{\ell_{10}} \in [0,n), \forall \pi_{\ell_{12}} \in [0,n), \\ \qquad \underbrace{*[\mathit{malloc}_{\ell_6}(\lambda_{\ell_4} \mapsto \pi_{\ell_{10}}, \bot) + \pi_{\ell_{12}}]}_{\texttt{mat2}[\pi_{\ell_{10}}][\pi_{\ell_{12}}]} = \texttt{mat1}[\pi_{\ell_{12}}][\pi_{\ell_{10}}] \end{cases}$$

To detail the property, we have :

- both `mat1` and `mat2` are arrays (of size `n`, which is given by a different component of the abstract memory state),
- they both contain, in each of their cells pointers to fresh arrays (again of size `n`) with $*[\mathit{malloc}_{\ell_1}(\bot) + \pi_{\ell_4}]$ that is just $\texttt{mat1}[\pi_{\ell_4}]$ when we unfold the definition of `mat1`,
- then each cell of the matrix `mat2` contains the value of the corresponding cell in the matrix `mat1` that has been transposed, with $*[\mathit{malloc}_{\ell_6}(\lambda_{\ell_4} \mapsto \pi_{\ell_{10}}, \bot) + \pi_{\ell_{12}}]$ that corresponds to $\texttt{mat2}[\pi_{\ell_{10}}][\pi_{\ell_{12}}]$ after unfolding the definition of `mat2` and the one of $*[\mathit{malloc}_{\ell_2}(\bot) + \pi_{\ell_{10}}]$,
- the values of `i` and `j` are not described in the property above because, at the end of the loop, if $\texttt{n} < 0$ their values would be 0 and not `n`.

6.2 Bootstrap of the paging in an x86 operating system

The program introduced in Fig. 10 is an excerpt of an x86 operating system, in particular the bootstrap of the expanded memory. In this context, we consider an array `page_directory` of size 1024, where the entry at index `virtual_page` holds the value $\texttt{physical\_page} \times 4096 + \texttt{flags}$. Each entry of this array, as a first approximation, is used to map the 4 KiB memory region starting at virtual address $4096 \times \texttt{page\_virtual}$ to the corresponding 4 KiB region beginning at physical address `physical_page`, with some extra parameters defined by $\texttt{flags} \in [0, 4096)$. Then, setting up `page_directory` allows to describe the first 4 MiB of the virtual memory (each page being 4 KiB long). In Fig. 10, all the pages of the first 4 MiB are mapped such that each virtual address matches its physical address. However, the flags of the memory regions differ. The first 8 pages (32 KiB) are reserved to the BIOS and the bootloader with flag 1 (superviser-only, read-only). The code of the operating system is then placed (by the bootloader) just after. It is of size `os_code_pages` that is only known at link-time, and the flag for this region is 5 (user-accessible, read-only). Then, all the remaining virtual addresses of the first 4 MiB are given as free space to the

```
1 int os_code_pages;
2 uint32_t page_directory[1024];
3 int page = 0;
4 while (page < 8) {
5   page_directory[page] = page * 0x1000 +
      ↪ 0b001;
6   current_page++;
7 }
8 int i = 0;
9 while (i < os_code_pages) {
10  page_directory[page] = page * 0x1000 +
      ↪ 0b101;
11  page++;
12  i++;
13 }
14 while(page < 1024) {
15  page_directory[page] = page * 0x1000 +
      ↪ 0b111;
16  page++;
17 }
```

```
ℓ1  page_directory := malloc(1024);
ℓ2  page := 0;
ℓ3  while page < 8 do
ℓ4    page_directory[page] := page * 4096 + 1;
ℓ5    page := page + 1
ℓ6  done;
ℓ7  i := 0;
ℓ8  while i < os_code_pages do
ℓ9    page_directory[page] := page * 4096 + 5;
ℓ10   page := page + 1;
ℓ11   i := i + 1
ℓ12 done;
ℓ13 while page < 1024 do
ℓ14   page_directory[page] := page * 4096 + 7;
ℓ15   page := page + 1
ℓ16 done ℓ17
```

Fig. 10. Paging bootstrap in an x86 operating system

operating system with flag 7 (user-accessible, read/write). In reality, the memory layout would be more complex (in particular, we need to deal with more range of addresses that are reserved), but this example is representative of what one would find in an operating system.

Analyzing such programs requires to be precise about the content of the paging structures in order to prove that memory accesses are legitimate. In particular, techniques like array smashing [3], or dynamic partitioning of arrays [8,10,6] do not provide enough relations between the indexes of arrays and their contents.

The abstract domain presented in this paper is able to prove that, at the end of the program execution, and if we assume that $\texttt{os\_code\_pages} < 1024$, the following properties about the assignments hold:

$$
\begin{cases}
\texttt{page\_directory} = \mathit{malloc}_{\ell_1}(\bot) \wedge \\
\forall \pi_{\ell_3} \in [0, 8), \underbrace{*[\mathit{malloc}_{\ell_1}(\bot) + \pi_{\ell_3}]}_{\texttt{page\_directory}[\pi_{\ell_3}]} = 4096 \times \pi_{\ell_3} + 1 \wedge \\
\forall \pi_{\ell_8} \in [0, \texttt{os\_code\_pages}), \underbrace{*[\mathit{malloc}_{\ell_1}(\bot) + \pi_{\ell_8} + 8]}_{\texttt{page\_directory}[8+\pi_{\ell_8}]} = 4096 \times \pi_{\ell_8} + 32773 \wedge \\
\forall \pi_{\ell_{13}} \in [0, 1016 - \texttt{os\_code\_pages}), \\
\quad \underbrace{*[\mathit{malloc}_{\ell_1}(\bot) + \pi_{\ell_{13}} + \texttt{os\_code\_pages} + 8]}_{\texttt{page\_directory}[8+\texttt{os\_code\_pages}+\pi_{\ell_{13}}]} = \\
\qquad\qquad 4096 \times \texttt{os\_code\_pages} + 4096 \times \pi_{\ell_{13}} + 32775 \wedge \\
\texttt{i} = \texttt{os\_code\_pages} \wedge \\
\texttt{page} = 1024
\end{cases}
$$

These properties can be understood as follows:

- `page_directory` is an array (of size `1024`, which is given by a different component of the abstract memory state),
- the cell at offset $i \in [0, 8)$ contains the value $4096 \times i + 1$,
- the cell at offset $i \in [8, \texttt{os\_code\_pages} + 8)$ contains the value $4096 \times i + 5$, which is translated into: the cell at offset $j \in [0, \texttt{os\_code\_pages})$ contains the value $4096 \times j + 8 \times 4096 + 5$,
- the cell at offset $i \in [\texttt{os\_code\_pages}+8, 1024)$ contains the value $4096 \times i + 7$, which is translated into: the cell at offset $j \in [0, 1024 - \texttt{os\_code\_pages} - 8)$ contains the value $4096 \times j + 4096 \times \texttt{os\_code\_pages} + 8 \times 4096 + 7$,
- the value of `i` is `os_code_pages` and the one of `page` is 1024.

6.3 Index of the first element greater than a given value

```
1 int n, x;
2 int *arr = malloc(n * sizeof(int));
3 int result;
4 if (arr[n-1] <= x) {
5   result = n-1;
6 } else {
7   result = 0;
8   while (arr[result] <= x) {
9     result++;
10  }
11 }
```

```
ℓ1  arr := malloc(n);
ℓ2  if arr[n − 1] ≤ x then
ℓ3    result := n − 1
ℓ4  else
ℓ5    result := 0;
ℓ6    while arr[result] ≤ x do
ℓ7      result := result + 1
ℓ8    done
ℓ9  endif ℓ10
```

Fig. 11. Index of the first element greater than a given value in a (sorted) array

We consider the program introduced in Fig. 11 that, given an integer `x` and a sorted array `arr` of size `n` (supposed greater than 0), returns the index of the first element of `arr` which is greater than `x`, or `n-1` in case no element is greater. Note that, even if `arr` is not sorted, no out-of-bound accesses are performed. The properties that are computed by the domain and presented below are also useful to analyze linear interpolations of tabulated functions. Indeed, it is required to prove that the point in which a function is interpolated (here x) is between the indexes of two consecutive tabulated keys in order to prove that the result of the interpolation is between the corresponding tabulated values.

The abstract domain is able to express the following loop invariant at program label ℓ_6:

$$\begin{cases} \texttt{arr} = \mathit{malloc}_{\ell_1}(\bot) \wedge \\ \texttt{arr}[\texttt{n} - 1] > \texttt{x} \wedge \\ \forall \pi_{\ell_6} \in [0, \lambda_{\ell_6}), \texttt{arr}[\pi_{\ell_6}] \leq \texttt{x} \wedge \\ \forall \pi_{\ell_6} \in \{ \lambda_{\ell_6} \}, \texttt{result} = \pi_{\ell_6} \end{cases}$$

with $\lambda_{\ell_6} \in [0, \mathtt{n})$, thanks to the widening threshold (described in Sect. 5) that is used because of `arr` is of size `n`. The reason why λ_{ℓ_6} cannot take the value of `n` is because the second line of the invariant contradicts the fourth line. This contradiction is found by using a heuristic that, when applying $\mathit{filter}^\sharp_{\ell_6,\mathtt{t}}$, checks whether the negation appears in the still-effective comparison predicates represented by the $\mathit{guards}^\sharp$ element of the abstract memory state.

Then, after exiting the loop at program label ℓ_9 we get the following property:

$$\begin{cases} \mathtt{arr} = \mathit{malloc}_{\ell_1}(\bot) \wedge \\ \mathtt{arr}[\mathtt{n}-1] > \mathtt{x} \wedge \\ \forall \pi_{\ell_6} \in [0, \mathtt{result}), \mathtt{arr}[\pi_{\ell_6}] \leq \mathtt{x} \wedge \\ \forall \pi_{\ell_6} \in \{\, \mathtt{result} \,\}, \mathtt{arr}[\pi_{\ell_6}] > \mathtt{x} \end{cases}$$

The two last lines of the property indicate that `result` is the index of the first element that is greater than `x`.

7 Implementation details and limits

As mentioned in the introduction, the abstract domain presented in the article has been implemented in the `Astrée`© static analyzer. This section provides some implementation details and discusses the main limitations of the abstract domain.

Representation of partial functions. Partial functions are widely used in the abstract memory state. In the implementation, those functions are represented by functional maps implemented as balanced binary search trees. This allows to access or update elements in logarithmic time in the number of elements (that is always bounded by the number of program labels times the number of nested loops).

Interprocedural analysis. As `Astrée`© performs a polyvariant analysis of procedures (equivalent to a call by copy), like the one described in [2], the abstract domain has been quickly adapted to support multiple procedures' analysis. The solution was to associate to each program label a *path* described as a stack of *tokens*, described in [14]. This method corresponds to applying a partitioning over the traces, and the *tokens* characterize an element of the partition of the traces at specified control points. Thus, it is possible to use the manual directives provided by the partitioning abstract domain to fine-tune the analysis.

Forward goto. Such instructions are supported in the implementation by using a conservative approach: we apply the abstract transfer functions that correspond to the loops exited and entered between the source and destination of the `goto` statement within the AST.

Complex types C programs may include complex elements like casts and struct or union types. Because the abstract domain presented in this paper requires handling pointers as left-values, it was not possible to fully leverage the abstraction layer provided by the *struct* domain described in [15]. At this time, retrieving the value of a previously assigned left-value is possible only when the types trivially match. In particular, it is currently not possible to recompose an expression from multiple assignments. It would be an interesting addition in the future.

Memoization. When an abstract expression has to be dereferenced and a matching assignment is found, the identifier of the assignment is stored in a mutable field of the abstract expression so that, if it is later dereferenced again, the memoized assignment will be checked before going through the usual process.

Types that cannot be represented Some expressions, as floats, cannot be represented by abstract expressions. Still, in the implementation, $rval^\sharp$ can either represent an integer/pointer (as in this paper), but also the dereference of a pointer in a specified type. It is then possible, for example, to copy an array of floats and keep the property that the two arrays contain the same values.

Using information of other domains. In order to enhance the expressiveness of the abstract domain, we use properties inferred by the other domains to refine the abstract memory state. In particular, for each dereference that appears in an abstract expression that is in the abstract state, we add to it its abstract value by using only the non-relational value domains. The properties computed by the interval domain [5], the congruence domain [9] and the bitfield domain [17] are particularly useful to search efficiently which assignments and guards could be affected by a new assignment. The properties inferred by the Andersen's pointer analysis [1] are also used to reduce the loss of information when assigning a left-value which cannot be translated into an abstract left-value, or which cannot be proved within the bounds of its base.

Sharing information to other domains. In order to be beneficial to the analysis, sharing discovered properties to other domains is essential. When a right-value associated to an assignment or a guard is computed thanks to universal eliminations, or when the value of a loop counter is fixed when exiting a loop, then the information is propagated to the other domains. In addition, when an alarm (division by 0, integer overflow, out-of-bound access, …) was raised by another domain and is proved false in this domain, then it is removed.

8 Conclusion

In this paper, we proposed new abstractions for proving programs that manipulate memory blocks by folding loops — that is — by universally quantifying assignments and comparison predicates. In the case of assignments, quantification is used both for the left and the right values. This typically allows the

right-value to be expressed in terms of the left-value. This technique has proven effective for analyzing expanded memory bootstrap in operating systems when it relies on parameters unknown at analysis time. The abstract expressions used in the abstract memory state are affine forms in the loop counter values and in symbolic dereferences. The latest are unfolded lazily, allowing affine forms in loop counter values to be inferred. In the case where no precise information can be inferred by the abstract domain about the left-value of an assignment, other abstract domains can limit the loss of precision. The abstract domain has been implemented the `Astrée`© static analyzer, and we provided insights on how to move from the idealized abstract domain presented in the paper to an implementation in a real-word C static analyzer.

Future Work. In the short term, we plan to integrate this work into Mopsa [12], a modular open platform for static analysis. Looking ahead, as `Astrée`© targets mostly critical embedded code, it does not need to support recursive functions and backward `goto` that are generally prohibited. However, it would be interesting to test the effectiveness of the abstract domain when adding recursion counters that count the number of recursive calls. Another challenge would be to merge this work with the one presented in [4] to rewrite expressions that contain both pointer dereferences and implicit/explicit type conversions. Lastly, we are interested in potential applications of the techniques described in the paper to prove properties about the content of intrusive linked-lists.

Acknowledgements We deeply thank the anonymous referees for the precise reviews. We also thank Marco Campion and Guannan Wei for the insightful conversations.

References

1. Andersen, L., Institut, K.U.D.: Program Analysis and Specialization for the C Programming Language. DIKU rapport, Datalogisk Institut, Københavns Universitet (1994)
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation, Complexity, Analysis, Transformation. Springer (2002). https://doi.org/10.1007/3-540-36377-7_5
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation. ACM (2003). https://doi.org/10.1145/781131.781153
4. Boillot, J., Feret, J.: Symbolic transformation of expressions in modular arithmetic. In: Hermenegildo, M.V., Morales, J.F. (eds.) Static Analysis. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-44245-2_6
5. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: International Symposium on Programming. Dunod (1976). https://doi.org/10.1145/390019.808314
6. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926399
7. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_38
8. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1040305.1040333
9. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics (1989). https://doi.org/10.1080/00207168908803778
10. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Programming Language Design and Implementation. ACM (2008). https://doi.org/10.1145/1375581.1375623
11. Journault, M., Miné, A.: Static analysis by abstract interpretation of the functional correctness of matrix manipulating programs. In: Rival, X. (ed.) Static Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_13
12. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Verified Software. Theories, Tools, and Experiments. Springer (2020). https://doi.org/10.1007/978-3-030-41600-3_1
13. Liu, J., Rival, X.: An array content static analysis based on non-contiguous partitions. Computer Languages, Systems & Structures **47** (2017). https://doi.org/10.1016/j.cl.2016.01.005, special issue on the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2015)

14. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Proceedings of the 14th European Conference on Programming Languages and Systems. ESOP'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_2
15. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. SIGPLAN Not. **41**(7) (Jun 2006). https://doi.org/10.1145/1159974.1134659
16. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Verification, Model Checking, and Abstract Interpretation. Springer (2006). https://doi.org/10.1007/11609773_23
17. Miné, A.: Abstract domains for bit-level machine integer and floating-point operations. In: Fleuriot, J., Höfner, P., McIver, A., Smaill, A. (eds.) ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation. EPiC Series in Computing, vol. 17. EasyChair (2013). https://doi.org/10.29007/b63g
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
19. Venet, A.: The gauge domain: Scalable analysis of linear inequality invariants. In: Computer Aided Verification. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_15