Relational abstract interpretation of arrays in assembly code

Clément Ballabriga, Julien Forget, Jordy Ruiz

Noname manuscript No.
(will be inserted by the editor)

Relational abstract interpretation of arrays in assembly code

Clément Ballabriga · Julien Forget · Jordy Ruiz

the date of receipt and acceptance should be inserted later

Abstract In this paper, we propose a static analysis technique for assembly code, based on abstract interpretation, to discover properties on arrays. Considering assembly code rather than source code has important advantages: we do not require to make assumptions on the compiler behaviour, and we can handle closed-source programs. The main disadvantage however, is that information about source code variables and their types, in particular about arrays, is unavailable. Instead, the binary code reasons about data-locations (registers and memory addresses) and their sizes in bytes.

Without any knowledge of the source code, our analysis infers which sets of memory addresses correspond to arrays, and establishes properties on these addresses and their contents. The underlying abstract domain is relational, meaning that we can infer relations between variables of the domain. As a consequence, we can infer properties on arrays whose start address and size are defined with respect to variables of the domain, and thus can be unknown statically. Currently, no other tool operating at the assembly or binary level can infer such properties.

Keywords Abstract Interpretation; Array Analysis; Assembly Code; Polyhedra

1 Introduction

Assembly code analysis offers high fidelity reasoning about software behaviour. Combined with assembly code reconstruction from binary code (e.g. with IDAPro [15],

J. Forget · C. Ballabriga
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
E-mail: firstname.lastname@univ-lille.fr

J. Ruiz
Imperial College London, UK
E-mail: j.ruiz@imperial.ac.uk

angr [36], or OTAWA [3] in our case), it enables the analysis of software whose source code is unavailable, such as third-party libraries, legacy programs, or malware. This has a wide range of applications, including reverse engineering, binary code rewriting, binary code reuse, vulnerability detection, Worst-Case Execution Time analysis, and more [10]. However, one major challenge is that information on program *variables* and their *types* is unavailable in the assembly code. Instead, we must reason at a lower level of abstraction, and consider *data-locations*, that is to say registers and memory addresses, and their *size*.

In this paper, we propose a static analysis of assembly code, based on abstract interpretation, and focus on discovering properties on data-locations that correspond to arrays. More precisely, since arrays are source-code concepts, we aim at identifying iterative accesses to evenly spaced addresses and establishing global properties that hold for the content at all of these addresses.

1.1 Motivating example

We use the example of Figure 1 to illustrate our contribution. Even though we provide the source code to ease the presentation, recall that we operate on the assembly code instead. The example analyses the header of IP packets, which have variable length due to the possible presence of options. In the figure, upper-case identifiers correspond to constants. The instruction STATIC_ASSERT(b) ends the analysis and reports an error if b does not hold at the program location where this instruction appears. The main program (Figure 1a) fills the packet, checks the content of its header, then sends the packet. Function check_header checks the header options. Each option consists of four bytes, the first of which is the option type. The function checks that options have valid types according to the standard (i.e. less than MAX_OPT), and rejects options LSR and SSR, which are widely considered unsafe. It also verifies that the header ends with EOOL. Even though this last check is not mandatory, this illustrates how we can handle "end-of-array" values (experiments in Section 9 show a similar example for non-null terminated string check). A second program fills the header (Figure 1b), and in particular the header length, its options, and sets the end of header byte (EOOL). We assume that all assigned options are below LSR (this is simulated by a random value below LSR). Note that the exact number of options is unknown statically, but is assumed to be less than the maximum allowed by the standard (again, this is simulated by a random value).

Our analysis establishes that all the assertions of function check_header hold (note that `LSR` $<$ `SSR` $<$ `MAX_OPT`), based solely on the program compiled code. In order to come to this conclusion, the analysis first discovers that addresses written inside the loop at l6-9 of function fill_header form an array, whose cells all have values lower than LSR. Later, it detects that the loop at l5-10 of function check_header reads from the same array and thus proves that the assertions always hold. Existing static analyses for binary/assembly code fail to establish such a property due to the unknown header size. Actually, they do not consider any cell of this unknown-length array as a memory location of interest, and thus do not track their content. As a side note, since our analysis of check_header is performed statically, and since the analysis proves that assertions always hold, the call to check_header can be safely removed from the program, because it is dead code.

```
void check_header(uint8_t* packet) {
  // slight simplification, assume an 8 bits size
  uint8_t i,last_opt_pos=packet[0];

  for (i = OPTIONS_START; i < last_opt_pos; i++) {
    STATIC_ASSERT(packet[i*4] != LSR);
    STATIC_ASSERT(packet[i*4] != SSR);
    STATIC_ASSERT(packet[i*4] != EOOL);
    STATIC_ASSERT(packet[i*4] < MAX_OPT);
  }
  // this is actually not mandatory
  STATIC_ASSERT(packet[last_opt_pos*4] == EOOL);
}

int main() {
  char pkt[PACKET_SIZE];
  fill_header(pkt)
  fill_payload(pkt);
  check_header(pkt);
  send_pkt(pkt);

  return 0;
}
```

(a) Main program

```
void fill_header(uint8_t *packet) {
  // each option is 4 bytes
  uint8_t i, last_opt_pos = OPTIONS_START+getRand(MIN_NB_OPT, MAX_NB_OPT);
  packet[0] = last_opt_pos;
  ... // assign bytes 1->19
  for (i = OPTIONS_START; i < last_opt_pos; i++) {
    packet[i*4] = getRand(1, LSR-1);
    ... // assign bytes i*4+1->i*4+3
  }
  packet[last_opt_pos*4] = EOOL;
}
```

(b) Header initialization

Fig. 1: IP packet verification

1.2 Contribution

Our approach has two main benefits. First, we compactly represent constraints on arrays, in a way that does not depend on the array size. Compared to approaches that analyze separately the content at each cell address in an array, this reduces the complexity and helps keeping the analysis tractable. Second, and perhaps more importantly, our approach supports arrays whose range is unknown statically, either due to a statically unknown start address or a statically unknown size. To the best of our knowledge, other existing binary/assembly-level analyses fail to establish *any* property on addresses that belong to such statically unknown address ranges. Furthermore, our analysis is fully automated and can thus be applied directly to closed-source programs. To summarize, our contributions are the following:

- A new abstract domain, called POLYMA$_R$P, to represent properties on arrays in an assembly program;
- An abstract interpretation procedure that computes the abstract state of an assembly program at each location of the program. The procedure automatically identifies arrays and infers global properties on their content;
- Experiments that show the capabilities and limitations of our work.

2 Overview

In this section we provide an overview of the main challenges of our work and of how they are addressed on our motivational example. Recall that we focus on two objectives: identifying arrays, and representing their properties compactly. In order to achieve these two objectives, we combine and extend several existing techniques and abstract domains:

- *POLYMAP* [4] is an abstract domain for representing properties of assembly programs, which maps data locations to abstract state variables. POLYMAP is *relational*, it supports relations between addresses (and their contents) that are unknown statically. In this work we adapt POLYMAP so that it can map address ranges (arrays) to abstract state variables. Relying on a relational domain is key to handle arrays whose range is unknown statically;
- *Domain Summarisation* [16] is a technique for abstract interpretation of arrays, which associates the content of all the cells of an array to a single abstract state variable that summarizes global properties on the cell contents. In this work we adapt summarisation, originally designed for representing source code states, to assembly code;
- *Circular Linear Progressions* [33] represent discrete sets of evenly spaced values. We adapt this model to represent the set of addresses of an array. While CLPs are defined using constant values, instead in our adapted *Symbolic Linear Progressions* (SLP) model, address ranges are defined by variables of the abstract state (a variable for the start address, a variable for the number of elements, and a constant element size);
- *Avatars* [25] provide an abstract representation for optional numerical values. We adapt avatars to represent optional arrays, that is to say arrays that are potentially empty (that may have size zero). This enables to accurately and efficiently handle constraints related to array assignments in loops, when analyzing the first loop steps, where we must combine (join) properties between an empty array and a partially assigned one.

As an example, let us consider the program of Figure 1 and the analysis of the loop l6-9 in function fill_header:

- The analysis notices several stores at evenly spaced addresses (at l7) and thus creates a *SLP* to represent this set of addresses. The SLP, which we denote $b|^4_n$, represents the array assigned inside the loop. In this notation, b is the start address of the array, 4 the distance in bytes between each cell and n the number of cells. At the loop exit, we have the constraints $b = \texttt{OPTIONS\_START}$ and $\texttt{MIN\_NB\_OPT} \leq n \leq \texttt{MAX\_NB\_OPT}$:
- We use a single *summary* variable x_c to represent the content of all the array cells. At the loop exit, we obtain the constraint $1 \leq x_c \leq \texttt{LSR}$;

- *POLYMAP* tracks that, at the end of the loop, the content of any address in range $b|_n^4$ is represented by variable x_c;
- *Avatars* come in handy to establish the constraints on x_c at l6. Note that when $i = 0$ (first iteration of the loop), $n = 0$, so $b|_n^4$ represents an empty array, and variable x_c represents the content of no address. Avatars enable us to handle this "empty" case, and to establish the following constraint at the end of the loop, without relying on costly disjunctions of constraints:

$$\begin{cases} 1 \leq x_c < \texttt{LSR} & \text{if } n > 0 \\ x_c \text{ undefined} & \text{otherwise} \end{cases}$$

We conclude this overview by discussing the current limitations of our work (see Section 9 for more detailed examples):

- The first limitation concerns the type of properties that we can infer on array contents. As an example, assume that fill_header assigns an option `01` lower than LSR in some array cell, and an option `02` greater than LSR in another array cell. Then, we would obtain $\texttt{01} \leq x_c \leq \texttt{02}$ and we would not be able to prove that the assertion on LSR in function check_header holds. This is due to the approximations of the underlying numerical domain (Polyhedra [13], which only support linear constraints), combined with the fact that we only infer global array properties (summarisation);
- The second limitation concerns the address stores that we can identify as array assignments. Testing that a set of stores writes to an evenly spaced set of addresses is actually a difficult problem, so we rely on heuristics (see Section 6 for details). As a consequence, we sometimes fail to identify arrays (e.g. with triangular nested loops). Let us emphasize however that failing to identify some arrays still yields a sound analysis: we will miss some valid properties, but we will never infer invalid properties. Similarly, identifying a set of addresses as an array, while it is not declared as such in the source code, remains sound;
- The third limitation concerns scalability. There remains a lot of room for improving the scalability of our approach, by adapting various optimization techniques developed for source code abstract interpretation over the past decades. Still, our experiments show that the concepts developed in this paper can be used to analyze properties that are currently out of the scope of any other existing tool, so we believe that this work is an important step towards abstract interpretation of assembly programs of larger scale.

3 Related works

Below, we summarize previous works related to the two main issues tackled in our work: identifying data-locations of interest, in particular arrays, and abstract interpretation of arrays. Lots of efforts have been applied to the recovery of information about the variables of the source code from which the binary code originates. An important milestone is the *Value Set Analysis* (VSA) [2], which tries to identify addresses corresponding to variables, and also infers an approximation of their possible values, using abstract interpretation. Variations of VSA using different abstract domains have been proposed [23, 5], and/or incorporated in binary code analysis tools, such as CodeSurfer [1], BAP [8] and Jakstab [23]. VSA relies on a non-relational domain, which limits the kind of

addresses that can be identified as variables (typically only constant addresses, or addresses expressed as a constant offset to the stack pointer). In [34], authors use a hierarchy of abstract domains, some of which are relational, but relational constraints are ultimately transformed into non-relational ones using constant propagation and/or approximation. Finally, the POLYMAP domain of [4] is relational, but it does not provide support for array analysis.

Binary code type inference consists in recovering the types of the source code variables [10]. In particular, *data structure identification* focuses on the identification of aggregate types, such as records and arrays, by analysing how a program accesses memory. *Abstract Structure Identification* (ASI) [31] is an early work on this topic targeted for the analysis of Cobol programs. It takes an approach opposite to our work, by progressively decomposing aggregates (the set of aggregates is known at the start of the analysis) into simpler components, based on accesses to addresses located inside the address range of an aggregate. ASI was combined with VSA to tackle data structure identification in binary code [32]. Similar techniques were pursued, but using dynamic analysis in [14,37]. Instead, as in our analysis, Troshina et al. [38] detect arrays by analysing memory accesses that use affine expressions, i.e. accesses to addresses of the form $a_i = b + i * s$. Compared with this area of research (except VSA+ASI), we not only infer constraints on the address and types of source variables, but also on their contents.

Many static analysis techniques have been proposed to analyse array properties in source code, including decision procedures for dedicated logics [7,20], theorem proving [24,22] and abstract interpretation. A crucial difference with our work is that, since these analysis at targetted for source code they do not address the problem of identifying arrays. We do however reuse some existing techniques that have been proposed for abstract interpretation of arrays in source code, and adapt them for the analysis of assembly code. Abstract interpretation of arrays usually relies on *summarizing* a collection of variables corresponding to the content of different cells of an array with an auxiliary variable [6,16]. The initial technique was extended to enable partitioning array addresses into several symbolic intervals, and assigning different summary variables to each partition [18]. Summarisation usually relies on *array smashing*, which computes a property for each summary variable such that the property holds for all the summarized variables, meaning that only global properties of array contents (or slices thereof), are discovered. The discovery of properties that relate the contents of different cells was studied in [21,12]. Non-contiguous partitions have also been studied in [26]. In our work, we rely on summary variables and array smashing, and we do not support array segmentation.

Circular Linear Progressions were originally introduced to represent the *content* of data-locations in binary code [33]. We use a similar formalism, which we call Symbolic Linear Progressions, to represent address ranges. SLPs are defined with respect to variables of our abstract domain (i.e. the start address and number of elements can be a variable), as opposed to constants in CLPs.

4 A simple assembly language

To simplify the presentation, we consider programs of the minimalist assembly language `MEMP` from [4] (Polymalys actually supports the more complex ARM A32 instruction set). Reconstruction of the assembly code starting from binary code is out of the

Programs	::=	$l_1 : I_1, l_2 : I_2, \ldots, l_n :$ `END`
Labels	::=	$\{l_1, l_2, \ldots\}$
Registers	::=	$\{r_1, r_2, \ldots\}$
Constants	::=	$\{c_1, c_2, \ldots\}$
Instructions	::=	
$r_1 \leftarrow \mathrm{OP}^c(r_2, r_3)$	|	`OP r1 r2 r3`
$r \leftarrow c$	|	`SET r c`
Emulate undefined r	|	`RAND r`
$r_1 \leftarrow *(r_2)$	|	`LOAD r1 r2`
$*(r_1) \leftarrow r_2$	|	`STORE r1 r2`
Branch to l if $r \blacklozenge 0$ (with $\blacklozenge \in \{<, =, >\}$)	|	`BR`$\blacklozenge$ `r l`
Halt	|	`END`

Fig. 2: Syntax of `MEMP`

$$\frac{}{(\texttt{SET r c}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r : c], *)} \qquad \frac{c = random()}{(\texttt{RAND r}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r : c], *)}$$

$$\frac{\mathcal{R}(r_2) = c_2 \qquad \mathcal{R}(r_3) = c_3 \qquad c_1 = \mathrm{OP}^c(c_2, c_3)}{(\texttt{OP r1 r2 r3}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r_1 : c_1], *)}$$

$$\frac{\mathcal{R}(r_2) = c_2 \qquad *(c_2) = c_1}{(\texttt{LOAD r1 r2}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}[r_1 : c_1], *)}$$

$$\frac{\mathcal{R}(r_1) = c_1 \qquad \mathcal{R}(r_2) = c_2}{(\texttt{STORE r1 r2}, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}, *[c_1 : c_2])}$$

$$\frac{P[pc] = \texttt{BR}\blacklozenge\ \texttt{r l} \qquad \mathcal{R}(r) \neg \blacklozenge 0}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{c} (pc + 1, \mathcal{R}, *)} \qquad \frac{P[pc] = \texttt{BR}\blacklozenge\ \texttt{r l} \qquad \mathcal{R}(r) \blacklozenge 0}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{c} (l, \mathcal{R}, *)}$$

$$\frac{P[pc] = \texttt{END}}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{e} (\mathcal{R}, *)} \qquad \frac{P[pc] = I \qquad I \notin \{\texttt{END}, \texttt{BR}\blacklozenge\} \qquad (I, \mathcal{R}, *) \xrightarrow{i} (\mathcal{R}', *')}{P \vdash (pc, \mathcal{R}, *) \xrightarrow{c} (pc + 1, \mathcal{R}', *')}$$

Fig. 3: Semantics of `MEMP`.

scope of this paper and can be performed using existing tools (our prototype relies on OTAWA [3] for that purpose). `MEMP` makes the following simplifying assumptions: data accesses are always of the same size, memory accesses are aligned to the word size and are values in $\mathbb{Z}$, there are no integer overflows, and function calls are inlined (these limitations could be lifted using for instance [9,35]). Its syntax is detailed in Figure 2. OP^c denotes the concrete semantics of operation OP. $*(r)$ denotes the content at the address specified by r.

Figure 3 details the small-steps semantics of `MEMP`. Function $\xrightarrow{i}$ specifies the semantics of data-flow operations. It operates in a context $(\mathcal{R}, *)$, where $\mathcal{R}$ maps registers (from the set of processor registers R) to their values (in $\mathbb{Z}$), and $*$ maps memory addresses (in $\mathbb{Z}$) to their values (in $\mathbb{Z}$). Both mappings are assumed to be initially empty. Given a mapping m, term $m[x : y]$ denotes a mapping identical to m, except that x is mapped to y. Function $\xrightarrow{c}$ specifies the semantics of control-flow operations. It adds a program counter pc to the previous context. $\xrightarrow{e}$ denotes the last transition of the program.

5 Abstract domain

Our abstract domain relies on several previous formalisms. In this section, we first briefly recall these existing formalisms and briefly explain what we use them for (Sections 5.1, 5.2), and then introduce our own abstract domain POLYMA$_R$P (Section 5.3).

5.1 POLYMAP: a relational domain for assembly code analysis

In this section, we recall the main definitions related to the POLYMAP domain, which was introduced in [4]. Though POLYMAP relies on Polyhedra, it can however easily be adapted to rely on a different relational domain, e.g. Octagons, to reduce complexity, as long as that domain supports equality constraints between two variables of the abstract domain. We recall the main definitions of POLYMAP below.

Let $|S|$ denote the cardinality of set S. Let C_n denote the set of linear constraints in $\mathbb{Z}^n$ on a set of n variables taken in some set $\mathcal{V}$. We denote $\langle c_1, c_2, ..., c_m \rangle$ the polyhedron p consisting of all the vectors in $\mathbb{Z}^n$ that satisfy constraints c_1, c_2, ..., c_m (with $c_i \in C_n$, $1 \leq i \leq m$). We denote $card(p) = n$ the number of dimensions of p. In the rest of the paper, the term *variable* implicitly refers to polyhedron variables (a.k.a the domain dimensions). This should not be confused with source code variables, which are never considered. Instead, we analyse the contents of *data-locations*, that is to say of registers and memory locations accessed by the program. We denote:

- $\mathcal{P}$ the set of polyhedra;
- $s \in p$ when s (with $s \in \mathbb{Z}^{card(p)}$) satisfies the constraints of polyhedron p;
- $p \sqsubseteq_\diamond p'$ iff $\forall s \in p, s \in p'$;
- $p'' = p \sqcup_\diamond p'$ the *convex hull* of p and p';
- $p'' = p \sqcap_\diamond p'$ the union of the constraints of p and p';
- $vars(p)$ the set of variables of p, where $|vars(p)| = card(p)$ by definition;
- $proj(p, x_1 \dots x_k)$ the projection of p on space $x_1 \dots x_k$, with $k < |card(p)|$;
- $cyl(p, x)$ the cylindrification of p by x, as defined in [28] (which basically removes x from the constraints of p);
- $p[x_i/x_j]$ the substitution of variable x_j by x_i in p, which first applies $cyl(p, x_i)$ and then substitutes x_i for x_j in the remaining polyhedra constraints (or does nothing when $x_i = x_j$);
- We say that "c holds for p" when $p \sqsubseteq_\diamond \langle c \rangle$.

An abstract state in POLYMAP [4] is a triple $(p, \mathcal{R}^\sharp, *^\sharp)$. The polyhedron p specifies the constraints on the variables of the abstract state. The *register mapping* $\mathcal{R}^\sharp$ maps registers to variables. The *address mapping* $*^\sharp$ maps address variables to content variables. These mappings evolve during the analysis, because the polyhedra variables are not known when starting the analysis. Instead they are created/removed as the analysis progresses, and so associations are added/removed from the mappings.

Example 1 Consider the following abstract state of POLYMAP:

$$(\langle x_2 = x_0, x_3 = x_1, x_0 = 4, x_1 \geq 5 \rangle, \{r_0 : x_0, r_1 : x_1\}, \{x_2 : x_3\})$$

Registers r_0, r_1, are respectively mapped to variables x_0, x_1. The content of the address represented by x_2 is represented by x_3. Polyhedra constraints state that memory address 4 ($x_2 = x_0 = 4$) contains a value greater than 5 ($x_3 = x_1 \geq 5$).

5.2 Compact array representation

5.2.1 Summarisation

The technique for summarizing numerical domains introduced in [16] enables us to use a single variable to represent the content of multiple array cells. It relies on two summarisation operations, *fold* and *expand*, which we recall in this section (see [16] for more details). The *fold* operation takes two variables, and replaces them by a single variable that combines the constraints of both variables: $fold(p, x_1, x_2, z) = p[z/x_1] \sqcup_\diamond p[z/x_2]$.

Example 2 Let us consider a polyhedron with two variables x_1 and x_2, where $10 \leq x_1 \leq 20$ and $15 \leq x_2 \leq 25$. After applying operation $fold(p, x_1, x_2, z)$, these constraints are replaced by $10 \leq z \leq 25$. The variable z summarizes the possible values of the original variables x_1 and x_2.

In the following, we use this operation to fold the variables corresponding to different array cells into a single array content variable.

The *expand* operation is the reverse of the *fold* operation. It takes one variable and replaces it by two variables, such that each gets the same constraints as the original variable: $expand(p, z, x_1, x_2) = p[x_1/z] \sqcap p[x_2/z]$.

Example 3 Let us consider the output of the previous *fold* example. After applying $expand(p, z, x_3, x_4)$ we replace the variable z by x_3 and x_4, such that $10 \leq x_3 \leq 25$ and $10 \leq x_4 \leq 25$.

We will use this operation to expand an array content variable into different cell content variables. We extend the notation and let $expand(p, v, S)$ denote the expansion of variable v of polyhedron p to the set of variables S. Also, we have:

Property 1 (From [16]) For any polyhedron p, and variables x_1, x_2, and z, we have: $p \sqsubseteq_\diamond expand(fold(p, x_1, x_2, z), z, x_1, x_2)$

5.2.2 Symbolic Linear Progressions

We introduce *Symbolic Linear Progressions*, a representation adapted from *Circular Linear Progressions [33]*, to compactly represent address sets that correspond to arrays. A SLP is defined as a triple $b|_n^s \in \mathcal{V} \times \mathbb{N} \times \mathcal{V}$, where b is the *base* address (the address of the first array cell), s is the *step* (the distance between two adjacent cells), and n is the *count* (the number of cells). The set of addresses corresponding to a SLP is defined as: $b|_n^s := \{b + si, i \in [0, n[\}$. The base and count in the SLP are variables, but we only consider constant steps, because array elements size is in most programs fixed, and allowing variable steps would significantly complexify the analysis.

We define $last(x_1|_{n_1}^{s_1}) = x_1 + s_1 \cdot (n_1 - 1)$, representing the location of the greatest element in the SLP. We say that *slp* is a *singular access* iff its count can be proven to equal 1. In that case, the step has no purpose, it is replaced by $\perp$ (e.g. $x_1|_1^\perp$). We say that *slp* is *empty* iff its count can be statically proven to equal 0. Again, the step has no purpose in that case, it is replaced by $\perp$.

```
int main(void) {
  int x, y, i; int a[10];
  for (i = 0; i < 10; i++) {
    if (x > y) return;
    a[i] = getRand(x, y);
  } // For all 0<=k<10, x<=a[k]<=y
}
```

Fig. 4: Example of global array property.

5.2.3 Avatars

We rely on *avatars* [25] to analyse array-assignment loops accurately and efficiently. A well-known issue (see e.g. [29]) is that at the loop header we have to join constraints that consider the array to be completely unitialised with constraints that consider the array to be partially initialised: without proper care this could result in considering that the array may be completely uninitialised. Consider the example of Figure 4. Let $a|_n^4$ denote the SLP corresponding to the set of addresses of the array a. Let z denote the summarizing variable that corresponds to the content of a. We want to establish that when the program completes the loop we have $x \leq z \leq y$. Notice that at the beginning of the loop it does not hold that $x \leq y$ (when $i = 0$). However, at that program location, when $i = 0$, we also have $n = 0$, so the SLP denotes an empty array and variable z does not really represent any value. Actually, we can consider that $x \leq z \leq y$ also holds at the beginning of the loop, *provided that* z *exists*: if z does not exist, we cannot conclude that $x \leq y$. Following the *bi-avatar strategy* of [25], we replace z by a pair of *avatars* (z^-, z^+). It is replaced by z^- for lower inequalities and by z^+ for higher inequalities. So, the previous property is replaced by $x \leq z^+ \wedge z^- \leq y$. Then, we consider that z is defined iff $z^+ = z^-$. This means that, for a concrete state to belong to the concretisation of the abstract state, it must satisfy that if $n > 0$ then z^+ and z^- correspond to the same value in the concrete state (see Section 5.4 for more details). This technique enables us to accurately analyse the loop (see 7.2.2 for more details on avatar creation) and to conclude that we indeed have $x \leq z \leq y$ when the loop is complete. Also, splitting a variable into a pair of variables is far less costly than using disjunctions of constraints.

5.3 POLYMA$_R$P

Let R denote the set of registers, $\mathcal{V}$ denote the set of polyhedra variables. The set of abstract states of our abstract domain POLYMA$_R$P combines the previous formalisms and is denoted $\mathcal{A} = \mathcal{P} \times (R \nrightarrow \mathcal{V}) \times (\mathcal{V} \times \mathbb{N} \times \mathcal{V} \nrightarrow \mathcal{V} \cup (\mathcal{V} \times \mathcal{V}))$. An abstract state $a \in \mathcal{A}$, with $a = (p, \mathcal{R}^\sharp, *^\sharp)$, consists of a polyhedron p, a register mapping $\mathcal{R}^\sharp$ and an address mapping $*^\sharp$. The variable $\mathcal{R}^\sharp(r)$ represents the value of register r. Term $*^\sharp(b|_n^s)$ represents all the values at the addresses in the set $\{b + s.i, i \in [0, n[\}$. It can either be a single variable or a pair of avatar variables. We assume a predicate $isAv(x_b|_{x_c}^{step})$ that returns true in the former case, and false in the latter case.

Example 4 Consider the following abstract state of POLYMA$_R$P:

$$(\langle x_0 = 5, x_2 = x_0, x_3 \geq 1, x_4 > 0, x_5 = x_1, 0 \leq x_6 \leq 10, x_7^- \leq 0\rangle,$$
$$\{r_0 : x_0, r_1 : x_1\}, \{x_2|^1_{x_3} : x_4, x_5|^2_{x_6} : (x_7^+, x_7^-)\})$$

$x_2|^1_{x_3}$ represents an array starting at address 5 ($x_2 = x_0 = 5$), with at least one element ($x_3 \geq 1$), whose cells are separated by one byte (step equal to 1) and all have a positive value ($x_4 > 0$). The array represented by $x_5|^2_{x_6}$ has an unconstrained starting address. It is possibly empty ($0 \leq x_6 \leq 10$), and its content is represented by an avatar pair ((x_7^+, x_7^-)). The avatar constraints tell us that the array cells, if there are any, all have a negative value ($x_7^- \leq 0$).

In the following, in order to concisely define abstract state transformers (e.g. transfer functions) we use $(p', [r_i : x_i], [x_j : x_k])(\cdot)$ as a shorthand for $\lambda(p, \mathcal{R}^\sharp, *^\sharp).(p \sqcap_\diamond p', \mathcal{R}^\sharp[r_i : x_i], *^\sharp[x_j : x_k])$, and denote "$-$" when a state component remains unchanged. Whenever an unbound polyhedron variable appears in the lambda body, we implicitly assume that it is a fresh variable, that has never been used before during the analysis.

5.4 Concretisation

A concrete program state is a pair $(\mathcal{R}, *)$, where $\mathcal{R}$ maps registers to their content and $*$ maps addresses to their content. The set of concrete states is defined as $(R \nrightarrow \mathbb{Z}) \times (\mathbb{Z} \nrightarrow \mathbb{Z})$. A *valuation function* $\nu : \mathcal{V} \to \mathbb{Z}$ maps each polyhedron base or count variable to a value. Intuitively, a concrete state s belongs to the concretisation of an abstract state $s^\#$ iff we can find a valuation function such that the constraints of $s^\#$ are satisfied for s. For instance, in Figure 5, the concrete state s_1 belongs to the concretisation of the abstract state $s^\sharp$. Considering $\nu(x_5) = 1$, $\nu(x_6) = 2$, the SLP $x_5|^{x_6}_2$ corresponds to the set of addresses $\{1, 3\}$. We can then check that values of the data locations of s_1 satisfy all the constraints of the polyhedron of $s^\sharp$ (see below for consistency polyhedra). In this section, we define a concretisation function γ that maps an abstract state to the set of corresponding concrete states.

5.4.1 Expansion

The concretisation first expands SLP content variables into a separate variable for each address represented by the SLP. Indeed, even though the cells of an array must all respect the same constraints (imposed on the SLP content variable), they may still have different values, so the concretisation of each cell must be handled separately. We detail this expansion process below.

Let ν be a valuation function, p be a polyhedron and $b|^s_n$ be a SLP. Function *xpslp* expands the contents of a SLP as follows:

$$xpslp(\nu, p, b|^s_n) = p', S$$
$$S = \begin{cases} \text{a set of } \nu(n) \text{ fresh variables} & \text{if } \neg isAv(b|^s_n) \\ \text{a set of } \nu(n) \text{ fresh avatar pairs} & \text{otherwise} \end{cases}$$
$$p' = expand(p, *^\sharp(b|^s_n), S)$$

$$s^\sharp = (\langle x_0 = 5, 1 \leq x_1 \leq 2, x_5 = x_1, 0 \leq x_6 \leq 10, x_7^- \leq 0\rangle,$$
$$\{r_0 : x_0, r_1 : x_1\}, \{x_5|_{x_6}^2 : (x_7^+, x_7^-)\})$$

(a) Abstract state

$$s_1 = (\{r_0 : 5, r_1 : 1\}, \{1 : -1, 3 : -2\}) \in \gamma(s^\sharp) \quad (\nu(x_5) = 1, \nu(x_6) = 2)$$
$$s_2 = (\{r_0 : 5, r_1 : 2\}, \{2 : -4\}) \in \gamma(s^\sharp) \quad (\nu(x_5) = 2, \nu(x_6) = 1)$$
$$s_3 = (\{r_0 : 5, r_1 : 1\}, \{\}) \in \gamma(s^\sharp) \quad (\nu(x_6 = 0))$$

(b) Some concrete states in the concretisation of $s^\sharp$

$$\mathcal{C}_R = \langle x_0 = 5, x_1 = 1\rangle, \mathcal{C}_C = \langle x_5 = 1, x_6 = 2\rangle, \mathcal{C}_{A1} = \langle\rangle$$
$$\mathcal{C}_{A2} = \langle xp_0^+ = xp_0^- = *(\nu(x_5)) = -1, xp_1^+ = xp_1^- = *(\nu(x_5 + 2)) = -2\rangle$$
$$p' = \langle x_0 = 5, 1 \leq x_1 \leq 2, x_5 = x_1, 0 \leq x_6 \leq 10, xp_0^- \leq 0, xp_1^- \leq 0\rangle$$
$$xpvs(x_5|_{x_6}^2) = \{(xp_0^+, xp_0^-), (xp_1^+, xp_1^-)\}$$

(c) Consistency polyhedra for s_1, $\nu(x_5) = 1$, $\nu(x_6) = 2$

Fig. 5: Concretisation of an abstract state

Function *xpall* expands all the SLPs of an abstract state. It returns a new polyhedron, along with a function (denoted *xpvs* below) that maps SLPs to their corresponding expanded variable sets.

$$xpall((p, \mathcal{R}^\sharp, *^\sharp), \nu) = p', xpvs$$
$$\{b_1|_{n_1}^{s_1}, ..., b_n|_{n_n}^{s_n}\} = Dom(*^\sharp)$$
$$(p_1, S_1), ..., (p_n, S_n) = xpslp(\nu, p, b_1|_{n_1}^{s_1}), ..., xpslp(\nu, p, b_n|_{n_n}^{s_n})$$
$$p' = \prod_{i=1..n} p_i \qquad xpvs(b_i|_{n_i}^{s_i}) = S_i,\ \forall i = 1..n$$

We let $S[i]$ denote the i^{th} element in an ordered version of the expanded variable set S. Since all the expanded variables of a SLP are subject to the same constraints, the choice of the order is arbitrary.

5.4.2 Concretisation function

We decompose the set of constraints that must be satisfied by a concrete state to belong to the concretisation of an abstract state into several *consistency polyhedra*.

Definition 1 (Consistency polyhedra) Let $s^\sharp = (p, \mathcal{R}^\sharp, *^\sharp)$ be an abstract state and $s = (\mathcal{R}, *)$ be a concrete state:

$$\begin{aligned}
\mathcal{C}_R(\mathcal{R}^\sharp, \mathcal{R}) =& \langle \forall r \in Dom(\mathcal{R}^\sharp), \mathcal{R}^\sharp(r) = \mathcal{R}(r) \rangle \\
\mathcal{C}_C(*^\sharp, \nu) =& \langle \forall b|_n^s \in Dom(*^\sharp), b = \nu(b) \wedge n = \nu(n) \rangle \\
\mathcal{C}_{A1}(*, xpvs, \nu) =& \langle \forall b|_n^s \in Dom(xpvs) | \neg isAv(b|_n^s), \forall\ 0 \leq j < \nu(n) : \\
& xpvs(b|_n^s)[j] = *(\nu(b) + j \cdot s) \rangle \\
\mathcal{C}_{A2}(*, xpvs, \nu) =& \langle \forall b|_n^s \in Dom(xpvs) | isAv(b|_n^s), \forall\ 0 \leq j < \nu(n) : \\
& a_j^+ = a_j^- = *(\nu(b) + j \cdot s) \quad (\text{with } (a_j^+, a_j^-) = xpvs(b|_n^s)[j]) \rangle
\end{aligned}$$

$$\mathcal{C}((\mathcal{R}^\sharp, *^\sharp), (\mathcal{R}, *), xpvs, \nu) = \mathcal{C}_R(\mathcal{R}^\sharp, \mathcal{R}) \sqcap_\diamond \mathcal{C}_C(*^\sharp, \nu) \sqcap_\diamond \mathcal{C}_{A1}(*, xpvs, \nu) \sqcap_\diamond \mathcal{C}_{A2}(*, xpvs, \nu)$$

Intuitively, $\mathcal{C}_R$ requires concrete register values ($\mathcal{R}(r)$) to be equal to the corresponding variable in the abstract state ($\mathcal{R}^\sharp(r)$). $\mathcal{C}_C$ defines the constraints on the addresses accessed by the program: for each SLP, concrete values and abstract variables must be consistent for the base ($b = \nu(b)$) and the count of the SLP ($n = \nu(n)$). $\mathcal{C}_{A1}$ imposes constraints on the contents of the memory: expanded array cell content variables must be consistent with the corresponding concrete array cell content ($xpvs(b|_n^s)[j] = *(\nu(b) + j \cdot s)$). $\mathcal{C}_{A2}$ does the same in the case of array contents constrained by avatars: for non-empty arrays ($\nu(n) > 0$), the avatars of the pair must be equal ($a_j^+ = a_j^-$). If the array is empty however ($\nu(n) = 0$), the avatars of the pair remain unconstrained. $\mathcal{C}$ puts all the constraints together. The concretisation function γ is defined below.

Definition 2 The concretisation function $\gamma : \mathcal{A} \to \mathcal{P}(\mathcal{C})$ is defined as:

$$\gamma(s^\sharp) \coloneqq \{s | \exists \nu, (p', xpvs) = xpall(s^\sharp, \nu), \mathcal{C}((s^\sharp, s, xpvs, \nu) \sqcap_\diamond p' \neq \bot\}$$

In other words, to determine whether a concrete state s belongs to the concretisation of an abstract state $s^\sharp$, we build a polyhedron c that corresponds to the consistency constraints between $s^\sharp$ and s. If the intersection between c and the expanded polyhedron of $s^\sharp$ is not empty, then s is consistent with $s^\sharp$ and it belongs to the concretisation.

As an example, Figure 5 details the consistency polyhedra for $s^\sharp$, s_1, $\nu(x_5) = 1$, $\nu(x_6) = 2$. The expansion of the contents of SLP $x_5|_2^{x_6}$ yields two avatar pairs: (xp_0^+, xp_0^-) represents the content of the first address of the SLP, i.e. address 1 in s, while (xp_1^+, xp_1^-) represents the content of the second address, i.e. address 3 in s. The expansion also yields a modified polyhedron such that the constraints of the SLP content variable are reported to these two avatar pairs ($xp_0^- \leq 0, xp_1^- \leq 0$). The consistency polyhedra encode the correspondence between the data locations of s (registers r_0, r_1, addresses 1 and 3) and the variables of p'. Then s belongs to $\gamma(s^\sharp)$ because the constraints of the consistency polyhedra $\mathcal{C}_R$, $\mathcal{C}_C$, $\mathcal{C}_{A1}$, $\mathcal{C}_{A2}$ are compatible with those of p'.

6 Symbolic Linear Progressions

Before proceeding with the description of our abstract interpretation procedure, in this section we define a series of operations on Symbolic Linear Progressions that will be used by the procedure.

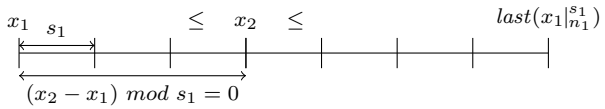

Fig. 6: x_2 falls within the range of $x_1|_{n_1}^{s_1}$.

6.1 Address in SLP range

We first consider the problem of testing whether some address falls within the address range of some SLP. This analysis is used to determine whether a load/store instruction accesses the content of a SLP tracked by the analysis. For the remainder of this subsection, we let $s^\sharp$ be an abstract state, $slp_1 = x_1|_{n_1}^{s_1}$ be a SLP of $s^\sharp$, x_2 be a variable of $s^\sharp$ corresponding to an address, and p be the polyhedron of $s^\sharp$. Figure 6 illustrates the conditions for x_2 to fall within the range of slp_1:

$$\text{The distance between } x_1 \text{ and } x_2 \text{ must be a multiple of } s_1 \quad (1)$$
$$x_2 \text{ must be located between the start and end of } slp_1 \quad (2)$$

Furthermore, we distinguish two cases of interest for our analysis: when slp_1 *must-contain* x_2 (for every point of p, x_2 is in slp_1), and when slp_1 *may-contain* x_2 (for some points of p, x_2 may be in slp_1).

6.1.1 Initial modulo offset

First, we consider how to test whether $x_2 - x_1$ is divisible by s_1 or not (for condition 2). To do so, we must first check whether there exists an integer v such that $(x_2 - x_1)\ mod\ s_1 = v$ holds for p. We call v the *initial modulo offset* between slp_1 and x_2. Unfortunately, the polyhedra domain does not support congruences. However, we can perform a sufficient (but not necessary) test by considering the following two cases of interest:

1. $(x_2 - x_1)$ has a single possible value;
2. $(x_2 - x_1)\ mod\ s_1$ has a single possible value. A sufficient (but not necessary) condition is that $x_2 = x_1 + n + s_1 \cdot x_k$ holds for some variable x_k for p. Candidates for x_k are the loop induction variables, because write addresses in arrays are usually a function of the loop iteration count.

Let p be a polyhedron, x_1, $x_2 \in vars(p)$ and $d \in \mathbb{N}$. Function *offset* computes the initial modulo offset as follows:

$$offset(p, x_1, x_2, d) = \begin{cases} n\ mod\ d & \text{if } \exists n \in \mathbb{Z}, p \sqsubseteq_\diamond \langle x_2 = x_1 + n \rangle \\ n'\ mod\ d & \text{if } \exists n' \in \mathbb{Z}, x_k \in vars(p) | p \sqsubseteq_\diamond \langle x_2 = x_1 + n' + d \cdot x_k \rangle \\ \top & \text{otherwise} \end{cases}$$

Example 5 In the following abstract state, we have $offset(p, x_1, x_2, 2) = 0$. Indeed, $x_2 = x_1 + 0 + 2 * x_3$, $0\ mod\ 2 = 0$.

$$(\langle x_2 = x_1 + 2 * x_3, n_1 \geq 8, 0 \leq x_3 \leq 4 \rangle, -, \{x_1|_{n_1}^2 : x_5, x_2 : x_6\})$$

6.1.2 Must contain

We denote $Must(slp_1, x_2)(s^\sharp)$ when slp_1 must-contain x_2 in abstract state $s^\sharp$. To test this property, we check that either conditions 2 and 1 both hold for p, or that $x_1|_{n_1}^{s_1}$ is a single address equal to x_2:

$$Must(x_1|_{n_1}^{s_1}, x_2)(p, \mathcal{R}^\sharp, *^\sharp) =$$
$$\begin{cases} p \sqsubseteq_\diamond \langle x_1 \leq x_2, last(x_1|_{n_1}^{s_1}) \geq x_2 \rangle \wedge (offset(p, x_1, x_2, s_1) = 0) & \text{if } s_1 \neq \bot \\ p \sqsubseteq_\diamond \langle x_1 = x_2, n_1 = 1 \rangle & \text{otherwise} \end{cases}$$

Example 6 In Example 5 we have $Must(x_1|_{n_1}^2, x_2)$, because the following holds:

- $s_1 = 2 \neq \bot$
- $offset(p, x_1, x_2, 2) = 0$
- $x_1 \leq x_2$
- $x_2 \leq x_1 + 2 * 4 \leq x_1 + 2 * 8 \leq last(x_1|_{n_1}^2)$

6.1.3 May contain

We denote $May(slp_1, x_2)(s^\sharp)$ when slp_1 may-contain x_2 in $s^\sharp$. First, the *MayOffset* auxiliary function checks the initial modulo offset. If the initial modulo offset is a multiple of s_1, or if we do not know whether it is or not, *MayOffset* returns an unconstrained polyhedron. Otherwise, the initial modulo offset is definitely not a multiple of s_1, so *MayOffset* returns an empty polyhedron.

$$MayOffset(p, x_1|_{n_1}^{s_1}, x_2) = \begin{cases} \langle\rangle \text{ if } offset(p, x_1, x_2, s_1) = 0 \\ \langle\rangle \text{ if } offset(p, x_1, x_2, s_1) = \top \\ \bot \text{ otherwise} \end{cases}$$

The May function is defined below. The main difference compared to the definition of *Must contain* is that we check that conditions 2 and 1 hold for *some* points of p, by checking that their intersection with p is non-empty.

$$May(x_1|_{n_1}^{s_1}, x_2)(p, \mathcal{R}^\sharp, *^\sharp) =$$
$$\begin{cases} (MayOffset(p, x_1|_{n_1}^{s_1}, x_2) \sqcap_\diamond \langle x_1 \leq x_2, last(x_1|_{n_1}^{s_1}) \geq x_2 \rangle \sqcap_\diamond p) \neq \bot \vee \\ \quad (\langle x_1 = x_2 \rangle \sqcap_\diamond p) \neq \bot & \text{if } s_1 \neq \bot \\ \langle x_1 = x_2 \rangle \sqcap_\diamond p \neq \bot & \text{otherwise} \end{cases}$$

Example 7 Let us consider the following abstract state (which is the same as in Example 5, except for the constraints on x_3).

$$(\langle x_2 = x_1 + 2 * x_3, n_1 \geq 8, 0 \leq x_3 \leq 9 \rangle, -, \{x_1|_{n_1}^2 : x_5, x_2 : x_6\})$$

We have $May(x_1|_{n_1}^2, x_2)$ (but we do not have $Must(x_1|_{n_1}^2, x_2)$) because:

- $s_1 = 2 \neq \bot$ holds for p
- $offset(p, x_1, x_2, 2) = 0$ holds for p
- $x_1 \leq x_2$ holds for p
- $x_2 \leq last(x_1|_{n_1}^2)$ holds only for a (non-empty) subpart of p: this is for instance true when $x_3 \leq 8$, but not when $x_3 = 9 \wedge n_1 = 8$.

6.2 Extending a SLP

In this section, we consider the problem of extending an existing SLP with an additional address. This operation is applied when an instruction stores at an address next to an existing SLP. To simplify the presentation, we only consider arrays accessed from start to end (i.e. with positive steps). First, we define function *istep*, which computes the step of a SLP when merging the first two addresses of the SLP. Let $max(p, x_2 - x_1)$ denote the maximum possible value of $x_2 - x_1$ in p:

$$istep(p, x_1|_{n_1}^{s_1}, x_2) = \begin{cases} m & \text{if } s_1 = \bot \wedge m \in \mathbb{N}_{>0} \wedge m \neq +\infty \\ s_1 & \text{otherwise} \end{cases}$$
$$\text{with } m = max(p, x_2 - x_1)$$

The heuristic SF is used when interpreting a store, to determine if an address, represented by a variable x_2, should be considered as a new address of an existing SLP $x_1|_{n_1}^{s_1}$. The *Merge* operation adds the address to the SLP. Both functions try to check that x_2 is located one step after the current end of $x_1|_{n_1}^{s_1}$ (i.e. that $x_1 + n_1 * s_1 = x_2$):

$$SF(x_1|_{n_1}^{s_1}, x_2)(p, \mathcal{R}^\sharp, *^\sharp) = \begin{cases} true & \text{if } p \sqsubseteq \langle x_1 + n_1 * s_1' = x_2 \rangle \\ false & \text{otherwise} \end{cases}$$
$$Merge(p, x_1|_{n_1}^{s_1}, x_2) = \begin{cases} x_1|_{n_1+1}^{s_1'} & \text{if } p \sqsubseteq \langle x_1 + n_1 * s_1' = x_2 \rangle \\ x_1|_{n_1}^{s_1} & \text{otherwise} \end{cases}$$
$$\text{with } s_1' = istep(p, x_1|_{n_1}^{s_1}, x_2)$$

Example 8 Let us consider the following abstract state:

$$(p, \mathcal{R}^\sharp, *^\sharp) = (\langle n_1 \geq 4, x_2 = x_1 + n_1 * 2, x_3 = 0, x_4 = 10 \rangle, -, \{x_1|_{n_1}^2 : x_3, x_2|_1^\bot : x_4\})$$

We have: $SF(x_1|_{n_1}^2, x_2)(p, \mathcal{R}^\sharp, *^\sharp) = true,\ Merge(p, x_1|_{n_1}^2, x_2) = x_1|_{n_1+1}^2$

Finally, the function *foldSLP* adds an address, represented by variable x_2, to an existing SLP $slp_1 = x_1|_{n_1}^{s_1}$, and updates the abstract state accordingly. It is applied only after checking for $SF(slp_1, x_2)$. We *Merge* the variable x_2 into the SLP slp_1. We also *fold* the SLP content and the content of x_2 into a single fresh variable x, then bind x to the (extended) SLP. The folded address (x_2) is mapped to a free, unconstrained, variable x_f.

$$\begin{aligned} foldSLP(slp_1, x_2)(p, \mathcal{R}^\sharp, *^\sharp) &= (p', \mathcal{R}^\sharp, *^{\sharp\prime}) \\ p' &= fold(p, *^\sharp(slp_1), *^\sharp(x_2|_1^\bot), x) \\ *^{\sharp\prime} &= *^\sharp[Merge(p, slp_1, x_2) : x, x_2|_1^\bot : x_f] \end{aligned}$$

Example 9 Function *foldSLP* is illustrated below on the abstract state of Example 8. We fold x_3 and x_4 into x_5, and remove unnecessary constraints:

$$\begin{aligned} &foldSLP(x_1|_{n_1}^2, x_2)(p, \mathcal{R}^\sharp, *^\sharp) = \\ &(\langle n_1 \geq 4, 0 < x_5 < 10, n_2 = n_1 + 1 \rangle, -, \{x_1|_{n_2}^2 : x_5\}) \end{aligned}$$

6.3 Avatars for empty SLPs

We rely on avatars to handle constraints on a SLP that is potentially empty. We assume a function *bac* (bi-avatar copy) that splits a variable into two avatars, following the bi-avatar strategy of [25][1]. We transpose the definition of the bi-avatar strategy to POLYMA$_R$P as follows:

Definition 3 Abstract state $(p, \mathcal{R}^\sharp, *^\sharp)$ follows the bi-avatar strategy iff, for all (x^-, x^+) in $Im(*^\sharp)$:

- No constraint of the form $ax^- \pm a_0x_0 \pm \cdots \pm a_nx_n \leq c$ appears in p;
- No constraint of the form $-ax^+ \pm a_0x_0 \pm \cdots \pm a_nx_n \leq c$ appears in p.

We overload the previous definitions of the summarisation operators *fold* and *expand* for avatars. In the remainder of the paper, we assume that abstract state transformers implicitly select the correct version of these operations depending on the type of their operands:

$$fold(p, (x_1^+, x_1^-), x_2, z) = p[z/x_1^+, z/x_1^-] \sqcup_\diamond p[z/x_2]$$
$$expand(p, (z^+, z^-), (x_1^+, x_1^-), x_2) = p[x_1^+/z^+, x_1^-/z^-] \sqcap_\diamond p[x_2/z^+, x_2/z^-]$$

fold is used when there is a STORE at an address next to an existing SLP: it combines the constraints on that address content (x_2) with the constraints on the SLP content ((x_1^+, x_1^-)) into a single variable (z). Note that we can safely replace the avatar pair by a single variable because the stored address will be merged into the SLP, so the resulting SLP is always non-empty. *expand* is used when there is a LOAD from an address inside a SLP: the constraints on the content of the SLP ((z^+, z^-)) are duplicated into the avatar pair ((x_1^+, x_1^-)), which will be mapped to the SLP content, and into a regular variable (x_2), which will be mapped to the load register.

7 Abstract interpretation

Our analysis proceeds by forward abstract interpretation [11], adapted to the analysis of programs of `MEMP` as described in [4]. It progressively computes the abstract state associated to each program label, using the abstract state transformers defined in the remainder of this section: the transfer function $(I)^\sharp$ of each instruction I, and the abstract domain operators (join $\sqcup$ and widening ∇). We let $M = Interpret(P)$ denote the result of the analysis of a program P of `MEMP`, where $M[l]$ denotes the abstract state obtained at the label l of P. The different abstract state transformers are illustrated throughout this section on the running example of Figure 7, whose behaviour is close to that of the C program of Figure 4. After an initialization step (l1-3), the program loops on l4-14. The loop index is represented by r_6, going from 0 to 10 (l3, 11-14). At each step of the loop, the value r_5 is between that of r_1 and r_2 (l6-9), and it is stored at address r_6 (l10), that is to say in the current array cell. l16 reads from cell 5 of the array. The analysis discovers that at the end of the loop addresses 0 to 10 belong to the same array and that their content is between r_1 and r_2.

[1] This simply consists in variable substitutions in the polyhedron of the abstract state.

```
1: RAND r1
2: RAND r2
3: SET r6 0
4: SUB r10 r1 r2
5: BR> r10 17
6: RAND r5
7: SUB r7 r2 r1
8: MOD r5 r5 r7
9: ADD r5 r5 r1
10: STORE r6 r5
11: ADD r6 r6 1
12: SET r12 10
13: SUB r11 r6 r12
14: BR< r11 4
15: SET r8 5
16: LOAD r9 r8
17: END
```

Label	*Polyhedron*	*Registers*	*Memory*
$4 \leftarrow 3$	$p_e = \langle x_6 = 0 \rangle$	$\mathcal{R}^\sharp_e = \{r_1 : x_1, r_2 : x_2, r_6 : x_6\}$	
6	$p_6 = p_e \sqcap \langle x_1 \leq x_2 \rangle$	$\mathcal{R}^\sharp_6 = \mathcal{R}^\sharp_e$	
10	$p_{10} = p_6 \sqcap \langle x_1 \leq x_5 \leq x_2 \rangle$	$\mathcal{R}^\sharp_{10} = \mathcal{R}^\sharp_6[r_5 : x_5]$	
13	$p_b = p_{10} \sqcap \langle x_9 = 1, x_8 = x_6 + 1, x_0 = x_6 \rangle$	$\mathcal{R}^\sharp_b = \mathcal{R}^\sharp_{10}[r_6 : x_8]$	$*^\sharp_b = \{x_0 \vert^\perp_{x_9} : x_5\}$
$unify(4 \leftarrow 3, 4 \leftarrow 13)$	$p_{b'} = \langle x_0 = 0, x_1 \leq x_5 \leq x_2, x_9 = 1, x_6 = 1 \rangle$	$\mathcal{R}^\sharp_e$	$*^\sharp_b$
$(avcp(p_e, p_b, x_0\vert^\perp_{x_9})).2$	$p_{b''} = cyl(p_{b'}, x_5) \sqcap \langle x_1 \leq x_5^-, x_5^+ \leq x_2 \rangle$	$\mathcal{R}^\sharp_e$	$*^\sharp_{b''} = \{x_0 \vert^\perp_{x_9} : (x_5^+, x_5^-)\}$
$(avcp(p_e, p_b, x_0\vert^\perp_{x_9})).1$	$p_{e'} = \langle x_0 = 0, x_1 \leq x_5^+, x_5^- \leq x_2, x_9 = 0, x_6 = 0 \rangle$	$\mathcal{R}^\sharp_e$	$*^\sharp_{b''}$
$(4 \leftarrow 3) \sqcup (4 \leftarrow 13)$	$p_j = \langle x_0 = 0, x_1 \leq x_5^+, x_5^- \leq x_2, x_9 = x_6, 0 \leq x_6 \leq 1 \rangle$	$\mathcal{R}^\sharp_e$	$*^\sharp_{b''}$
$10'$	$p_{10'} = p_j \sqcap \langle x_1 \leq x_{15} \leq x_2 \rangle$	$\mathcal{R}^\sharp_{10'} = \mathcal{R}^\sharp_e[r5 : x_{15}]$	$*^\sharp_{b''}$
$13'$ no-fold	$p_{13'} = p_{10'} \sqcap \langle x_{19} = 1, x_{17} = x_6 + 1, x_{18} = x_6 \rangle$	$\mathcal{R}^\sharp_{11'} = \mathcal{R}^\sharp_{10'}[r6 : x_{17}]$	$\{x_0 \vert^\perp_{x_9} : (x_5^+, x_5^-), x_{18} \vert^\perp_{x_{19}} : x_{15}\}$
$13'$ folded	$p_{13'} \sqcap_\diamond \langle x_{20} = x_9 + 1, x_1 \leq x_{15} \leq x_2 \rangle$	$\mathcal{R}^\sharp_{11'}$	$*^\sharp_{13'} = \{x_0 \vert^1_{x_{20}} : x_{15})\}$
16	$p_{16} = \langle x_1 \leq x_5^-, x_5^+ \leq x_2, x_6 = 10, x_9 = x_6, x_{21} = 5 \rangle$	$\mathcal{R}^\sharp_{16} = \mathcal{R}^\sharp_e[r_8 : x_{21}]$	$*^\sharp_{16} = \{x_0 \vert^1_{x_9} : (x_5^+, x_5^-)\}$
17	$p_{16} \sqcap_\diamond \langle x_1 \leq x_{22} \leq x_2 \rangle$	$\mathcal{R}^\sharp_{16}[r_9 : x_{22}]$	$*^\sharp_{16}$

Fig. 7: Running example of analysis

7.1 Transfer functions

This section details the transfer functions for the instructions of `MEMP`, that is to say how we compute the impact of an instruction on an abstract state.

7.1.1 Non-memory instructions

Transfer functions for instructions not directly related to memory manipulation remain unchanged compared to [4]:

$$(\texttt{OP r}_1\ \texttt{r}_2\ \texttt{r}_3)^\sharp = \begin{cases} (\langle x = \mathrm{OP}^c(\mathcal{R}^\sharp(r_2), \mathcal{R}^\sharp(r_3)) \rangle,\ [r_1 : x],\ -)(\cdot) & \text{if } linear(\mathrm{OP}^c) \\ (-,\ [r_1 : x],\ -)(\cdot) & \text{otherwise} \end{cases}$$

$$(\texttt{SET r}_1\ \texttt{c})^\sharp = (\langle x = c \rangle,\ [r_1 : x], -)(\cdot)$$

$$(\texttt{RAND r}_1)^\sharp = (-,\ [r_1 : x], -)(\cdot)$$

$$(\texttt{BR}\blacklozenge\ \texttt{r l})^\sharp = \begin{cases} (\langle \mathcal{R}^\sharp(r) \blacklozenge 0 \rangle,\ -,\ -)(\cdot) & \text{at } l \\ (\langle \neg(\mathcal{R}^\sharp(r) \blacklozenge 0) \rangle, -,\ -)(\cdot) & \text{at current label+1 (if } linear(\neg\blacklozenge)) \end{cases}$$

A binary operation binds the target register (r_1) to a variable (x) constrained to be equal to the combination of the variables bound to the operand registers ($\mathcal{R}^\sharp(r_2)$, $\mathcal{R}^\sharp(r_3)$). If the constraint cannot be expressed as a linear relation, x remains unconstrained. Concerning the branching instruction, the branching condition holds at the target label (l), while its negation holds at the next label (referred to as *filtering*). Transfer functions for $(SET)^\sharp$ and $(RAND)^\sharp$ are straightforward.

Example 10 In Figure 7 at label 4, just after analyzing label 3 ($4 \leftarrow 3$), registers r_1 and r_2 have been initialized by a RAND instruction, so they are mapped to unconstrained variables (x_1, x_2). Register r_6 has been SET to 0, it is mapped to x_6, which equals 0. At label 6, the negation of the branching condition of label 5 holds, so we add $x_1 \leq x_2$ to the previous constraints (as a simplification of $\mathcal{R}^\sharp(r_{10}) \leq 0 \wedge \mathcal{R}^\sharp(r_{10}) = \mathcal{R}^\sharp(r_1) - \mathcal{R}^\sharp(r_2)$).

7.1.2 Store

We first define several auxiliary functions that will be used to define $(STORE)^\sharp$. $SU(slp, r)$ operates a *strong update*: it updates the content of SLP *slp* to the content of register r. $WU(slp, r)$ operates a *weak update*: the content of *slp* may either remain unchanged or change to the content of r. Function $(STORE)^\sharp$ consists of the following operations:

- Strongly-update single-address SLPs that must-contain the store address (see function *SUAll*);
- Weakly-update SLPs that may-contain the address (see function *WUAll*);
- If no strong-update was performed, create a new SLP (see function *CR*);
- If needed, fold the new SLP into existing SLPs (see function *FoldAll*).

Strong and weak updates are defined as follows:

$$SU(slp, r) = ((\langle x = \mathcal{R}^\sharp(r)\rangle, -, [slp : x]))(\cdot)$$
$$WU(slp, r)(s) = \lambda s.s \sqcup SU(slp, r)(s)$$

When performing a weak-update, if $*^\sharp$ maps *slp* to a bi-avatar pair, then the free variable x introduced in the call to SU is split into an avatar pair, following the bi-avatar strategy [25]. This is however not necessary when performing a strong-update alone, because we only apply strong updates on singular-access, non-empty, SLPs (see *SUAll* below).

$SUAll(r_1, r_2)$ applies a strong update to all the SLPs whose count is 1 and which must-contain the address specified by register r_1. $WUAll(r_1, r_2)$ applies a weak update to all the SLP which may-contain that address.

$$SUAll(r_1, r_2) = \bigcirc_{x|_1^s | Must(x|_1^s, \mathcal{R}^\sharp(r_1))} SU(x|_1^s, r_2)$$
$$WUAll(r_1, r_2) = \bigcirc_{slp | May(slp, \mathcal{R}^\sharp(r_1))} WU(slp, r_2)$$

$CR(r_b, r_c, b, x_c)$ creates a new memory location mapping: the address is specified by register r_b, whose content is bound to variable b, and the content to store is specified

by register r_c, whose content is bound to variable x_c. In the memory mapping, we map the singular access $b|_s^\perp$ to x_c:

$$CR(r_b, r_c, b, x_c) = (\langle b = \mathcal{R}^\sharp(r_b), x_c = \mathcal{R}^\sharp(r_c), n = 1\rangle, -, [b|_n^\perp : x_c])(\cdot)$$

After creating a new memory location, represented by variable x, $FoldAll(x)$ tries to fold x with existing SLPs:

$$FoldAll(x) = \bigcirc_{slp|SF(slp,x)} foldSLP(slp, x)$$

The transfer function for store is finally defined as follows:

$$(\texttt{STORE r}_1\ \texttt{r}_2)^\sharp = \begin{cases} WUAll(r_1, r_2) \circ SUAll(r_1, r_2) & \text{if } \exists b|_1^s | Must(b|_1^s, \mathcal{R}^\sharp(r_1)) \\ FoldAll(x_1) \circ WUAll(r_1, r_2) \circ CR(r_1, r_2, x_1, x_2) & \text{otherwise} \end{cases}$$

Example 11 In Figure 7 the abstract state 13 shows the result of a *CR* (resulting from instruction 10). When analysing the STORE of label 10, no SLP must-contain the target address represented by x_6 (there are none so far), so we create a new SLP with a base address (x_0) equal to x_6, a count (x_9) equal to 1, and a step equal to $\perp$. The content variable of the SLP is equal to the content variable of r_5 (x_5).

Example 12 Instruction 13 is inside a loop, so we extend the previous SLP when analysing the second iteration of the loop ($13'$). At this iteration, the target address of the STORE (r_6) is 0 or 1 ($0 \leq x_6 \leq 1$). No existing SLP must contain it, so we first create SLP $x_{18}|_{x_{19}}^\perp$ (with $x_{18} = x_6$, $x_{19} = 1$, see label $13'$ no-fold). Then, we fold this SLP with $x_0|_{x_9}^\perp$, because $x_{18} = x_6 = x_9 = x_0 + x_9 * 1$, producing SLP $x_0|_{x_{20}}^1$ (with $x_{20} = x_9 + 1 = 2$, see label 13 folded). The content variables of the two SLPs ((x_5^+, x_5^-) and x_{15}) are folded into x_{15}, which becomes the content variable of the new SLP. x_{15} combines the constraints of the folded variables ($x_1 \leq x_{15} \leq x_2$, due to the $\sqcup_\diamond$ in *fold*).

7.1.3 Load

To compute the impact of a LOAD r1 r2, we must determine which SLP *slp* (there might actually be several) must-contain the load address ($\mathcal{R}^\sharp(r_2)$), as this implies that one of the addresses of *slp* is the load address. However, since the content of *slp* is summarized to a single variable x_c, it would be incorrect to say that the load destination register ($\mathcal{R}^\sharp(r_1)$, a.k.a x) is equal to x_c. Instead, we say that x has the same constraints as x_c (*expand* copies the constraints). Finally, for the case where x_c is an avatar pair, note that $Must$ ensures that *slp* is not empty.

$$(\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp = ((\bigsqcap_{x_c \in \{*^\sharp(slp) | Must(p, slp, \mathcal{R}^\sharp(r_2))\}} expand(p, x_c, x_c, x), -, -) \bigcirc (-, [r_1 : x], -))(\cdot)$$

Example 13 Figure 7 illustrates the result of a LOAD with the state at label 17. The source address, specified by r_8, is equal to 5 ($x_{21} = 5$), which must-be-contained by SLP $x_0|_{x_9}^1$ (because $x_0 = 0$ and $x_9 = x_6 \geq 10$). So we map the destination register r_9 to a new variable x_{22}, which has the same constraints as the content of the SLP. However, the address mapping and the constraints on the SLP content variable remain unchanged (because we *expand* (x_5^+, x_5^-) into itself and into x_{22}).

7.2 Abstract domain operators

This section details the definition of the join and widening operators. We first introduce two auxiliary operations *unify* and *avcp*. Operation $unify$ tries to map the data-locations of two different states to the same variables, in order to enable a more accurate states comparison. *avcp* creates an empty copy of a SLP and its constraints, to properly handle the analysis of the first iteration of a loop.

7.2.1 Unification

The correspondence between polyhedra variables and data locations is neither predefined nor fixed. Therefore, it may happen that two abstract states use different variables to designate the same data location. To enable a more accurate comparison of these two states, we must *unify* them first, which consists in trying to assign the same variables in the two states to the same data locations. The unification procedure is detailed in Algorithm 1. Function $matchVar(v_1, v_2, p_1, p_2)$ returns true if variable v_1 of p_1 is equivalent to variable v_2 of p_2[2]. *unify* replaces SLPs of $s_2^\sharp$ by their equivalent in $s_1^\sharp$ and does the same for register variables.

Algorithm 1 $unify(s_1^\sharp = (p_1, \mathcal{R}_1^\sharp, *_1^\sharp), s_2^\sharp = (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$

$(p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp'}) \leftarrow (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$
for all $(b_1|_{n_1}^s, b_2|_{n_2}^s) \in Dom(*_1^\sharp) \times Dom(*_2^\sharp)$ **do**
  if $matchVar(b_1, b_2, p_1, p_2) \wedge matchVar(n_1, n_2, p_1, p_2)$ **then**
    Replace b_2 by b_1, n_2 by n_1 and $*_2^\sharp(b_2|_{n_2}^s)$ by $*_1^\sharp(b_1|_{n_1}^s)$ in $(p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp'})$
  end if
end for
for all $r \in Dom(\mathcal{R}_1^\sharp) \cap Dom(\mathcal{R}_2^\sharp)$ **do**
  Replace $\mathcal{R}_2^\sharp(r)$ by $\mathcal{R}_1^\sharp(r)$ in $(p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp'})$
end for
return $(p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp'})$

Example 14 In Figure 7, $unify((4 \leftarrow 3, 4 \leftarrow 13))$ substitutes x_6 for x_8 in $\mathcal{R}_b^\sharp$, which becomes equivalent to $\mathcal{R}_e^\sharp$. Before doing the same substitution in p_b, we cylindrify x_6, so we remove constraint $x_6 = 0$ and replace constraints on x_0 and x_8 by $x_0 = 0$, $x_8 = 1$. Then the substitution transforms $x_8 = 1$ into $x_6 = 1$.

7.2.2 Empty SLP creation

We will now discuss how to handle SLPs that only appear in one of two states we wish to join. This is a key step to solving the problem of handling constraints on partially initialised arrays presented previously in Section 5.2.3. For instance, in the example of Figure 7, the abstract state for the loop back edge $(4 \leftarrow 13)$ contains a SLP $x_0|_{x_9}^\perp$ that

[2] A heuristic for this function is detailed in [4]. To summarize, to test $matchVar(v_1, v_2, p_1, p_2)$, we try to express v_1 and v_2 as linear expressions of a set of variables *npiv* chosen among the common variables of p_1 and p_2. The set *npiv* is determined using Gauss-Jordan elimination on the constraints of $p_1 \sqcup_\diamond p_2$

does not appear in the loop entry edge ($4 \leftarrow 3$). So, applying a join between the two edges directly would lose all constraints on the content variable of the SLP (x_5). To prevent this loss of precision we use the procedure *avcp* defined below.

Let $s_1^\sharp$, $s_2^\sharp$ be two abstract states, and $b|_n^s$ be a SLP that appears in $s_2^\sharp$ but not in $s_1^\sharp$. In that case, we propose to create an empty copy of $b|_n^s$ in $s_1^\sharp$, with the same constraints as in $s_2^\sharp$. Intuitively, this is sound because, as the copy is empty, any constraint on it will have no impact on the concretisation of $s_1^\sharp$. Let $cstr_x(p)$ denote the polyhedron p reduced to all the constraints on x. Let $copy(p_1, p_2, x) = cstr_x(proj(p_2, vars(p_1) \cup \{x\}))$, which extracts constraints on x that appear in p_2 and that are expressed in terms of variables of p_1. The procedure described in Algorithm 2 adds the empty copy of $b|_n^s$ from $(p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$ to $(p_1, \mathcal{R}_1^\sharp, *_1^\sharp)$. Remember that *bac* is the bi-avatar copy of Section 6.3.

Algorithm 2 $avcp((p_1, \mathcal{R}_1^\sharp, *_1^\sharp), (p_2, \mathcal{R}_2^\sharp, *_2^\sharp), b|_n^s)$

1: $x_c \leftarrow *_2^\sharp(x_b|_n^s)$
2: $p_2' \leftarrow bac(p_2, x_c, x_c^+, x_c^-)$ ▷ Split x_c according to the bi-avatar strategy
3: $p_1' \leftarrow p_1 \sqcap_\diamond \langle copy(p_1, p_2', x_b), copy(p_1, p_2', x_c^+), copy(p_1, p_2', x_c^-)\rangle$
4: $*_{2'}^\sharp \leftarrow *_2^\sharp[x_b|_n^s : (x_c^+, x_c^-)]$ ▷ Remap existing SLP to bi-avatar pair
5: $*_{1'}^\sharp \leftarrow *_1^\sharp[x_b|_0^\perp : (x_c^+, x_c^-)]$ ▷ Introduce new empty SLP
6: **return** $(p_1', \mathcal{R}_1^\sharp, *_{1'}^\sharp), (p_2', \mathcal{R}_2^\sharp, *_{2'}^\sharp)$

Example 15 In Figure 7, we decompose the impact of *avcp* on the loop back edge $(p_{b''}, *_{b'}^\sharp)$ and entry edge $(p_{e'})$, for SLP $x_0|_{x_9}^\perp$. Since *avcp* returns a pair $(s_1^\sharp, s_2^\sharp)$ we let $(s_1^\sharp, s_2^\sharp).1$ denote $s_1^\sharp$ and $(s_1^\sharp, s_2^\sharp).2$ denote $s_2^\sharp$ (e.g. in Figure 7). In the back edge, we split the content variable (x_5) into a pair of avatars $((x_5^+, x_5^-))$ and split its constraints between the avatars ($x_1 \leq x_5^-, x_5^+ \leq x_2$). We copy the SLP mapping into the memory mapping of the entry edge (which becomes $*_{b''}^\sharp$). We also copy into the entry edge the constraints of the back-edge that concern the SLP base variable (x_0) and the SLP content variable $((x_5^+, x_5^-))$. Finally, the count of the SLP is set to 0 ($x_9 = 0$) in the entry edge.

7.2.3 Join

The join procedure is described in Algorithm 3. The join procedure first unifies the two states to join, then adds in the first state the empty copies of the SLPs of the second state, joins the two polyhedra, and finally filters common data locations. Even though it would be sound to always add empty SLPs to the first state, this would dramatically increase the polyhedra dimensions, since we would never get rid of addresses that appear in only one of two states to join. Thus, we only perform empty SLP creation when either the join precedes a widening or when its count can be zero.

Example 16 In Figure 7, we show the result of the join of the entry and back-edge of label 4. After unification and empty SLP creation (which was detailed previously), the convex hull combines the constraints of the two edges on x_6 ($0 \leq x_6 \leq 1$) and x_9 ($x_9 = x_6$), and keeps the constraints on the avatar pair (x_5^+, x_5^-). Let us emphasize that

Algorithm 3 $(p_1, \mathcal{R}_1^\sharp, *_1^\sharp) \sqcup (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$

1: $(p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp\prime}) = unify((p_1, \mathcal{R}_1^\sharp, *_1^\sharp), (p_2, \mathcal{R}_2^\sharp, *_2^\sharp))$
2: **for all** $b_2|_{n_2}^{s_2} | b_2|_{n_2}^{s_2} \notin Dom(*_1^\sharp)$ **do**
3: **if** The join precedes a widening and $(p_2 \sqsubseteq_\diamond \langle n_2 > 0 \rangle$ **then**
4: $avcp((p_1, \mathcal{R}_1^\sharp, *_1^\sharp), (p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp\prime}), b_2|_{n_2}^{s_2})$
5: **else if** $(\langle n_2 = 0 \rangle \sqcap_\diamond p_2 \neq) \bot$ **then**
6: $avcp((p_1, \mathcal{R}_1^\sharp, *_1^\sharp), (p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp\prime}), b_2|_{n_2}^{s_2})$
7: **end if**
8: **end for**
9: $p \leftarrow p_1 \sqcup_\diamond p_2'$
10: **for all** $r \in Dom(\mathcal{R}_1^{\sharp'})$ **do**
11: **if** $\mathcal{R}_1^\sharp(r) = \mathcal{R}_2^{\sharp'}(r)$ **then** $\mathcal{R}^\sharp(r) \leftarrow \mathcal{R}_1^\sharp(r)$ **end if**
12: **end for**
13: **for all** $a \in Dom(*_1^{\sharp'})$ **do**
14: **if** $*_1^\sharp(a) = *_2^{\sharp'}(a)$ **then** $*^\sharp(a) \leftarrow *_1^\sharp(a)$ **end if**
15: **end for**
16: **return** $(p, \mathcal{R}^\sharp, *^\sharp)$

constraints on $*^\sharp(x_0|_{x_9}^\bot)$ would have been lost if we had not applied a *avcp* before the convex hull. This illustrates how we are able to avoid losing constraints on a partially initialised array.

7.2.4 Widening

The widening operator ∇ is used to ensure that the analysis reaches a fix-point when analysing loops. The widening procedure is described in Algorithm 4. It first unifies the two states, then widens the first polyhedron, and finally filters common data locations.

Algorithm 4 $(p_1, \mathcal{R}_1^\sharp, *_1^\sharp) \nabla (p_2, \mathcal{R}_2^\sharp, *_2^\sharp)$

1: $(p_2', \mathcal{R}_2^{\sharp'}, *_2^{\sharp\prime}) = unify((p_1, \mathcal{R}_1^\sharp, *_1^\sharp), (p_2, \mathcal{R}_2^\sharp, *_2^\sharp))$
2: $p \leftarrow p_1 \nabla_\diamond p_2'$
3: **for all** $r \in Dom(\mathcal{R}_1^{\sharp'})$ **do**
4: **if** $\mathcal{R}_1^\sharp(r) = \mathcal{R}_2^{\sharp'}(r)$ **then** $\mathcal{R}^\sharp(r) \leftarrow \mathcal{R}_1^\sharp(r)$ **end if**
5: **end for**
6: **for all** $a \in Dom(*_1^{\sharp'})$ **do**
7: **if** $*_1^\sharp(a) = *_2^{\sharp'}(a)$ **then** $*^\sharp(a) \leftarrow *_1^\sharp(a)$ **end if**
8: **end for**
9: **return** $(p, \mathcal{R}^\sharp, *^\sharp)$

Example 17 Figure 7 shows the result of the widening at label 16. The constraints on x_6 were first widened to $0 \leq x_6 \leq 10$, using lookahead narrowing [17] (note that these constraints do not appear at label 13/13' because widening has not been applied yet). Then, because the negation of the test of the BR at label 14 holds at label 16, we have the constraint $x_6 \geq 10$ (filtering). Combining the constraints, we obtain $x_6 = 10$.

7.2.5 Inclusion

To determine when the analysis reaches a fix-point, we must test abstract state inclusion: the fix-point is reached when, for all program labels, the analysis computes an abstract state that is included in the abstract state computed previously at that label. Let $s_1^\sharp = (p_1, \mathcal{R}^\sharp{}_1, *^\sharp{}_1)$ and $s_2^\sharp = (p_2, \mathcal{R}^\sharp{}_2, *^\sharp{}_2)$. The inclusion operator $\sqsubseteq^\sharp$ is defined as follows:

$$s_1^\sharp \sqsubseteq s_2^\sharp \Leftrightarrow p_1' \sqsubseteq_\diamond p_2 \wedge \mathcal{R}_2^\sharp \subseteq \mathcal{R}_1^{\sharp'} \wedge *_2^\sharp \subseteq *_1^{\sharp'}$$
$$\text{with } (p_1{}', \mathcal{R}_1^{\sharp'}, *^{\sharp'}) = \mathit{unify}(s_2^\sharp, s_1^\sharp)$$

8 Soundness

The following theorem establishes the soundness of our analysis. It states that the analysis always computes an over-approximation of the possible concrete states of the program under analysis.

Theorem 1 *Let P be a* MEMP *program. Let $M = \mathit{Interpret}(P)$. Then, for any concrete state s_{init}: $(P \vdash (l_1, s_{init}) \xrightarrow{c}{}^*(\ell, s)) \Longrightarrow (s \in \gamma(M[\ell]))$*

The theorem follows from the soundness of the abstract domain operators and of the transfer functions. The soundness of the abstract domain operators *unify*, ∇, $\sqcup$, and of the transfer functions for non-memory instructions directly follows from the proofs previously established in [4]. In the rest of this section we focus on the soundness of our avatar strategy, and on the soundness of transfer functions for LOAD and STORE.

8.1 Avatars

As stated in [25], for the analysis to be sound, the avatar strategy must respect the *independence* property, which basically says that dropping the constraints on an avatar has no impact on the other variables. [25] proved that the bi-avatar strategy satisfies the independence property. So we just establish that our analysis only produces states that respect the bi-avatar strategy:

Lemma 1 *Let P be a program of MEMP and $M = \mathit{Interpret}(P)$. Then:*

$$\forall l \in \mathit{Dom}(M), M[l] \textit{ follows the bi-avatar strategy}$$

Proof Most abstract state transformers have no impact on avatar constraints, in particular most transfer functions operate on registers, not memory. The only exceptions are *WU* (as part of $(STORE)^\sharp$) and *avcp*. They both add constraints following the bi-avatar strategy: when we replace a variable z by an avatar pair (z^+, z^-), in the polyhedron constraints, we replace z by z^- in lower inequalities and by z^+ in higher inequalities. □

The following lemma states that function *avcp* is sound.

Lemma 2 *Let a_1,$a_2 \in \mathcal{A}$, $b|_n^s \in Dom(*_2^\sharp)$. Let $(a_1', a_2') = avcp(a_1, a_2, b|_n^s)$. We have: $\gamma(a_1) \subseteq \gamma(a_1') \wedge \gamma(a_2) \subseteq \gamma(a_2')$.*

Proof $\gamma(a_2) \subseteq \gamma(a_2')$ follows from the soundness of the bi-avatar strategy. For $\gamma(a_1) \subseteq \gamma(a_1')$, just note that the consistency predicate $\mathcal{C}_{A2}$ will be the same for a_1' and a_1, as the count of the SLP in a_1' is 0. So both states have the same concretisation. □

8.2 Memory instructions

Assume s_1 is a concrete state before some instruction and s_2 is the concrete state after applying that instruction to s_1. Then our interpretation procedure is sound iff: $s_1 \in \gamma(s_1^\sharp) \Rightarrow s_2 \in \gamma(s_2^\sharp)$. In this section we prove that this property holds for `LOAD` and `STORE`. To do so, we consider the constraints added by the corresponding transfer function in $s_2^\sharp$. We detail how this translates into the consistency polyhedron of $s_2^\sharp$. Then we prove that the changes between the consistency polyhedra of $s_1^\sharp$ and $s_2^\sharp$ match with the concrete semantics of the instruction. We start with an auxiliary property, that will be used to prove the soundness of $(\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp$.

Property 2 Let p be a non-empty polyhedron. Let x_1 be a variable that is free in p. Then, $p \sqcap_\diamond p[x_1/x_2] \neq \bot$

Proof Trivial.

Then we state the soundness of $(\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp$:

Lemma 3 $(s_1 \in \gamma(s_1^\sharp) \wedge (\texttt{LOAD r}_1\ \texttt{r}_2, s_1) \xrightarrow{i} s_2) \Rightarrow s_2 \in \gamma((\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp(s_1^\sharp))$

Proof Let $s_1^\sharp = (p_1, \mathcal{R}_1^\sharp, *_1^\sharp)$, $s_1 = (\mathcal{R}_1, *_1)$, be such that $s_1 \in \gamma(s_1^\sharp)$. Let $s_2^\sharp = (p_2, \mathcal{R}_2^\sharp, *_2^\sharp) = (\texttt{LOAD r}_1\ \texttt{r}_2)^\sharp(s_1^\sharp)$, $(\texttt{LOAD r}_1\ \texttt{r}_2, s_1) \xrightarrow{i} s_2$, $s_2 = (\mathcal{R}_2, *_2)$, $\mathcal{R}_1^\sharp(r_2) = x_2$, $Must(s_1, x_1|_n^s, x_2)$. Since $s_1 \in \gamma(s_1^\sharp)$, there exists a valuation function ν such that $\langle \mathcal{C}(s_1^\sharp, s_1, xpvs_1, \nu)\rangle \sqcap_\diamond p_1' \neq \bot$, with $p_1', xpvs_1 = xpall(s_1, \nu)$. Let $p_2', xpvs_1 = xpall(s_2, \nu)$. Let $C_1 = \langle \mathcal{C}(s_1^\sharp, s_1, xpvs_1, \nu)\rangle$, $C_2 = \langle \mathcal{C}(s_2^\sharp, s_2, xpvs_1, \nu)\rangle$.

We have $C_2 = C_1 \sqcap_\diamond \langle \mathcal{R}_2^\sharp(r_1) = \mathcal{R}_2(r_1)\rangle$. Also, since $Must(s_1, x_1|_n^s, x_2)$, there exists j such that $0 \leq j < \nu(n)$, $\nu(x_1) + j \cdot s = \mathcal{R}_1(r2)$. Furthermore, $*_1(\mathcal{R}_1(r2)) = \mathcal{R}_2(r_1)$. Therefore, $C_1 \sqsubseteq_\diamond \langle xpvs_1(x_1|_n^s)[j] = \mathcal{R}_2(r_1)\rangle$. So $C_2 = C_1 \sqcap_\diamond C_1[\mathcal{R}_2^\sharp(r_1)/xpvs_1(x_1|_n^s)[j]]$.

Because of the *expand* in the `LOAD` transfer function, the set of constraints on $\mathcal{R}_2^\sharp(r_1)$ in p_2' is equivalent to the set of constraints on $xpvs_1(x_1|_n^s)[j]$ in p_1'. So $p_2' = p_1' \sqcap_\diamond p_1'[\mathcal{R}_2^\sharp(r_1)/xpvs_1(x_1|_n^s))[j]]$. Then $p_2' \sqcap_\diamond C_2 = p_1' \sqcap_\diamond C_1 \sqcap_\diamond (p_1' \sqcap_\diamond C_1)[\mathcal{R}_2^\sharp(r_1)/xpvs_1(x_1|_n^s))[j]]$. As $C_1 \sqcap_\diamond p_1' \neq \bot$, thanks to property 2, $C_2 \sqcap_\diamond p_2' \neq \bot$. The rest of the property follows. □

Now, we establish two auxiliary properties, that will be used to prove the soundness of $(\texttt{STORE r}_1\ \texttt{r}_2)^\sharp$.

Property 3 Let $s = (p, \mathcal{R}^\sharp, *^\sharp)$. Let $x_1|_n^s$, $x_2|_1^\bot \in Dom(*^\sharp)$. Then:

$$SF(x_1|_n^s, x_2)(s) \Rightarrow \gamma(s) \subseteq \gamma(foldSLP(x_1|_n^s, x_2, s))$$

Proof We prove that any concrete state that satisfies the consistency predicate for s also satisfies the predicate for $foldSLP(x_1|_n^s, x_2, s)$. Let ν be a valuation function for s and let $\nu' = \nu[n' \to \nu(n)+1]$. Let:

$$\begin{aligned}(p_1, \mathcal{R}_1^\sharp, *_1^\sharp) &= foldSLP(x_1|_n^s, x_2, s)\\ (p_2, \mathcal{R}_2^\sharp, *_2^\sharp), S_2 &= xpslp(\nu, p, x_2|_1^\perp)\\ (p_3, \mathcal{R}_3^\sharp, *_3^\sharp), S_3 &= xpslp(\nu, p_2, x_1|_n^s)\\ (p_4, \mathcal{R}_4^\sharp, *_4^\sharp), S_4 &= xpslp(\nu', p_1, x_1|_{n'}^s)\\ \{e_{\nu(n)+1}\} = S_2, \{e_1, \ldots, e_{\nu(n)}\} &= S_3, \{e_1, \ldots, e_{\nu(n)+1}\} = S_4\end{aligned}$$

We have $p_1 = fold(p, *^\sharp(x_1|_n^s), *^\sharp(x_2|_1^\perp), x)$.
Let $p_5 = expand(p_1, x, *^\sharp(x_1|_n^s), *^\sharp(x_2|_1^\perp))$. Assuming $SF(x_1|_n^s, x_2)(s)$ holds, we have $x_1|_{n'}^s = x_1|_n^s \cup \{x_2|_1^\perp\}$, so:

$$p_4 = expand(expand(p_5, *^\sharp(x_2|_1^\perp), \{e_{\nu(n)+1}\}), *^\sharp(x_1|_n^s), \{e_1, \ldots, e_{\nu(n)}\})$$
$$p_3 = expand(expand(p, *^\sharp(x_2|_1^\perp), \{e_{\nu(n)+1}\}), *^\sharp(x_1|_n^s), \{e_1, \ldots, e_{\nu(n)}\})$$

From Property 1, $p \sqsubseteq_\diamond p_5$. As *expand* is monotonous, we have $p_3 \sqsubseteq_\diamond p_4$.
Concerning $\mathcal{C}_{A1}$, assuming that $SF(x_1|_n^s, x_2)$ holds:

$$\langle \mathcal{C}_{A1}(*_4^\sharp, *, [x_1|_{n'}^s \to S_4], \nu')\rangle = \langle \forall 0 \le j < \nu'(n') : S_4[j] = *(\nu'(x_1) + j \cdot s)\rangle$$
$$\langle \mathcal{C}_{A1}(*_3^\sharp, *, [x_2|_1^\perp \to S_2], \nu)\rangle = \langle S_2[0] = *(\nu(x_2))\rangle = \langle S_4[\nu(n)] = *(\nu(x_1) + \nu(n) \cdot s)\rangle$$
$$\langle \mathcal{C}_{A1}(*_3^\sharp, *, [x_1|_n^s \to S_3], \nu)\rangle = \langle \forall 0 \le j < \nu(n) : S_4[j] = *(\nu(x_1) + j \cdot s)\rangle$$

So we have $\mathcal{C}_{A1}(*_3^\sharp, *, [x_2|_1^\perp \to S_2, x_1|_n^s \to S_3], \nu) = \mathcal{C}_{A1}(*_4^\sharp, *, [x_1|_{n'}^s \to S_4], \nu')$. A similar reasoning can be applied to $\mathcal{C}_{A_2}$. The rest of the property follows from the definition of the concretisation function. □

Property 4 Let $s_1^\sharp = (p_1, \mathcal{R}_1^\sharp, *_1^\sharp) \in \mathcal{A}$, and $r_1 \in Dom(\mathcal{R}_1^\sharp)$, such that there is a unique SLP $b|_n^s \in Dom(*_1^\sharp)$ with $May(b|_n^s, \mathcal{R}_1^\sharp(r_1))(s_1^\sharp)$. Then:

$$(s_1 \in \gamma(s_1^\sharp) \wedge (\texttt{STORE r}_1\ \texttt{r}_2, s_1) \xrightarrow{i} s_2) \Rightarrow s_2 \in \gamma(WU(b|_n^s, r_2)(s_1^\sharp))$$

Proof Let $s_1 = (\mathcal{R}_1, *_1)$, $s_2 = (\mathcal{R}_2, *_2)$, $s_2^\sharp = (p_2, \mathcal{R}_2^\sharp, *_2^\sharp) = WU(b|_n^s, r_2)(s_1^\sharp)$. Since $s_1 \in \gamma(s_1^\sharp)$, there exists a valuation function ν such that $\langle \mathcal{C}(s_1^\sharp, s_1, xpvs_1, \nu)\rangle \sqcap_\diamond p_1' \neq \bot$, with $p_1', xpvs_1 = xpall(s_1, \nu)$. Let $(p_2', xpvs_1) = xpall(s_2, \nu)$. Let $C_1 = \langle \mathcal{C}(s_1^\sharp, s_1, xpvs_1, \nu)\rangle$, $C_2 = \langle \mathcal{C}(s_2^\sharp, s_2, xpvs_1, \nu)\rangle$.

From the definition of WU, we have $s_2^\sharp = s_1^\sharp \sqcup SU(b|_n^s, r_2)(s_1^\sharp)$, it follows that $p_1' \sqsubseteq p_2'$.

Since $b|_n^s$ may overlap with $\mathcal{R}_1^\sharp(r_1)$, there may exists a j such that $0 \le j \le \nu(n)$ and $\mathcal{R}_1(r_1) = \nu(b) + s \cdot j$.

If j does not exist, then $C_1 = C_2$. Since $p_1' \sqsubseteq p_2'$, then $s_2 \in \gamma(s_2^\sharp)$.

If j does exist, let us make the following decompositions:
$C_1 = C \sqcap_\diamond \langle xpvs_1(b|_n^s)[j] = *_1(\nu(b) + j \cdot s)\rangle$, $C_2 = C \sqcap_\diamond \langle xpvs_1(b|_n^s)[j] = *_2(\nu(b) + j \cdot s)\rangle$. We have $C \sqsubseteq_\diamond \langle \mathcal{R}_1^\sharp(r_2) = \mathcal{R}_1(r_2)\rangle$, and from the definition of WU, $p_2' \sqsubseteq \langle xpvs_1(b|_n^s)[j] = \mathcal{R}_1^\sharp(r_2)\rangle$. It follows that:

$$\begin{aligned}(p_2' \sqcap_\diamond C) &\sqsubseteq \langle xpvs_1(b|_n^s)[j] = \mathcal{R}_1^\sharp(r_2)\rangle \sqsubseteq \langle xpvs_1(b|_n^s)[j] = \mathcal{R}_1(r_2)\rangle\\ &\sqsubseteq \langle xpvs_1(b|_n^s)[j] = *_2(\mathcal{R}_1(r_1))\rangle \sqsubseteq \langle xpvs_1(b|_n^s)[j] = *_2(\nu(b) + j \cdot s)\rangle\end{aligned}$$

Therefore, $p'_2 \sqcap_\diamond C_2 = p'_2 \sqcap_\diamond C$, which is not $\bot$ because $p'_1 \sqsubseteq p'_2$ and $p'_1 \sqcap_\diamond C \neq \bot$. □

Then we state the soundness of $(\texttt{STORE r}_1\ \texttt{r}_2)^\sharp$.

Lemma 4 $(s_1 \in \gamma(s_1^\sharp) \wedge (\texttt{STORE r}_1\ \texttt{r}_2, s_1) \xrightarrow{i} s_2) \Rightarrow s_2 \in \gamma((\texttt{STORE r}_1\ \texttt{r}_2)^\sharp(s_1^\sharp))$

Proof Follows from Property 3, Property 4. The soundness of CR and SU directly follows from [4]

9 Experiments

The abstract interpretation procedure presented previously is implemented in a prototype called Polymalys, as a plugin of OTAWA [3]. We rely on OTAWA for the assembly code reconstruction and control-flow analysis. In the following experiments, we first detail the capability of Polymalys in terms of array properties inference on several illustrative examples. Then, we provide performance metrics on programs from the Mälardalen and Polybench benchmarks.

9.1 Examples

From the tools considered in the related works, to the best of our knowledge, only CodeSonar, derived from CodeSurfer, supports the analysis of arrays. It does however fail to analyse the examples listed in this section, as well as the motivating example of Figure 1. This means that no tool operating at the assembly or binary level, other than Polymalys, can prove *any* of the assertions of the motivating example or of the programs presented in this section.

9.1.1 Global array properties

We start with examples that illustrate the benefits and limitations of inferring global properties on arrays.

Example 18 The following example produces a non-null terminated string, a common array bug. The program generates a random string, and uses strncpy() to copy it into an array that is too small to hold the terminating null byte. Note that the array size is unknown. Polymalys proves that the destination array does not contain any null byte.

```
void generate_string(char *t, int size) {
    int i;
    for (i = 0; i < size; i++)
      t[i] = getRand(1,255); // non-null byte
    t[i] = 0;
}

void string_copy(int size) {
    char src[size+1];
    generate_string(src, size);
    char dest[size];
```

```
    strncpy(dest, src, size);
    char x = getRand(0, size);
    STATIC_ASSERT(dest[x] != 0);
}
```

Example 19 The following example copies an array into another array. Polymalys correctly analyses that the two arrays have the same content.

```
1 void copy(void) {
2   char tab[10], tib[10], k; int i;
3   if ((k <= 0) || (k > 100)) return;
4   for (i = 0; i < 10; i++) tab[i] = k;
5   for (i = 0; i < 10; i++) tib[i] = tab[i];
6   for (i = 0; i < 10; i++) STATIC_ASSERT(tab[i] == tib[i]);
7 }
```

Example 20 In the program of Example 19, assume that we remove the loop at l4. Obviously, the assertion still holds. However, since we use array smashing, Polymalys cannot prove the assertion.

9.1.2 Array identification

This section illustrates the types of programs where Polymalys can, or cannot, identify arrays. First, we illustrate how programs with arrays whose addresses fall within overlapping intervals are handled.

Example 21 The following example manipulates an array of structures. Let x_{st} denote the address of *st*. Here, two SLPs are created: $b_0|^8_{n_0}$ and $b_1|^8_{n_1}$, where $b_0 = x_{st}$, $b_1 = x_{st} + 4$, $n_0 \geq 10$, $n_1 \geq 10$. Polymalys proves the assertion. It establishes that the two SLPs do not overlap, even though the intervals they cover do.

```
struct st { int a; int b;};
void array_struct(void) {
  int i,n; struct st st[n];
  if(n<10) return;
  for (i = 0; i < n; i++) st[i].a = 42;
  for (i = 0; i < n; i++) st[i].b = 51;
  STATIC_ASSERT(st[1].a==42 && st[5].b=51);
}
```

Example 22 The following example illustrates SLPs that do overlap. Here, three SLPs are created: one for each singular access tab[a], tab[b], and one for the whole array tab. Even though the array overlaps with the singular accesses, Polymalys tracks the contents of the three SLPs separately, and thus it can establish that the assertions never fail. Such an example would be difficult to support with an analysis based on array segmentation (e.g. [18,26]), because we do not know the relative ordering of *a* and *b*.

```
void overlap(void) {
  int tab[n], i, a, b;
  if(n<10) return;
```

```
  for (i = 0; i < n; i++) tab[i] = 42;
  if ((a >= 0) && (a < 10))
    if ((b >= 0) && (b < 10)) {
      tab[a] = 10; tab[b] = 10;
      STATIC_ASSERT(tab[a] == 10 && tab[b] == 10);
      STATIC_ASSERT(tab[3] >= 10 && tab[3] <= 42);
    }
}
```

Second, we consider multi-dimensional arrays.

Example 23 The following example fills a matrix with the same value in every cell. N and M are constants. A single SLP of size $N * M$ is created and Polymalys proves the assertion.

```
1 void matrix_fill(void) {
2   int mat[N][M],i,j,k;
3   for (i = 0; i < N; i++)
4     for (j = 0; j < M; j++) mat[i][j]=v;
5   k=getRand(0,N*M-1);
6   STATIC_ASSERT(mat[k]==v);
7 }
```

Example 24 In the program of Example 23, assume that N and M are both variables instead of constants. In that case Polymalys does not identify arrays. Indeed, this introduces non-linear constraints, which cannot be represented by Polyhedra.

Example 25 The following example contains triangular nested loops. Polymalys fails to identify any array, it is unable to identify sets of arrays of non-uniform sizes.

```
1 void triangular_fill(void) {
2   int mat[N][M],i,j,k;
3   for (i = 0; i < N; i++)
4     for (j = i; j < M; j++) mat[i][j]=v;
5 }
```

9.2 Performance

Experiments are performed on a PC with an Intel core i5 3470 at 3.2 Ghz, with 8 GB of RAM. Every benchmark has been compiled with ARM crosstool-NG 1.20.0 (gcc version 4.9.1), using the -O0 optimization level. Table 1 provides metrics for the following programs: examples from the previous section (overlap, struct_array, copy), the motivating example of Figure 1, a matrix multiplication program (matmult) from the Mälardalen benchmarks [19], and various programs (floyd−warshall, nussinov, covariance, correlation, gemver) from the Polybench benchmarks [30]. For each experiment, we provide the number of assembly instructions, the number of polyhedron variables and constraints, the time it takes for the analysis to complete (the preliminary steps performed by OTAWA are excluded from the measurements), the number of arrays in the program, and the number of arrays identified by Polymalys.

Execution times show that the analysis remains tractable for non-trivial programs, even though it increases steeply with the number of instructions. For bigger programs,

Bench	Instructions	Variables	Constraints	Time (ms)	Arrays	Found
overlap	82	32	36	237	1	1
struct_array	57	20	22	250	1	**2**
copy	88	29	30	341	2	2
motivating example	177	53	58	2602	1	1
matmult	213	70	72	24658	2	2
floyd−warshall	275	48	57	48213	1	1
nussinov	498	63	59	40090	2	2
covariance	1034	77	79	56106	4	**3**
correlation	807	94	90	99641	5	5
gemver	1021	124	123	116362	9	**8**

Table 1: Performance of Polymalys

we could replace the polyhedra domain by e.g. the octagon domain [27] to reduce the execution time, at the cost of over-approximation.

In the majority of cases, the number of detected arrays matches the actual number of arrays in the source code. Let us discuss experiments for which this is not the case. First, in the struct_array benchmark, we create two separate SLPs for each field of the structures in the array. This is a case where we detect "more arrays" than the source code actually contains. However, as detailed previously in Example 21, it is sound to handle the SLPs separately. In each of the covariance and gemver benchmarks, one array is missed. This is because the instruction writing in the array is within a triangular loop nest similar to Example 25.

10 Conclusion

We have presented an abstract interpretation procedure for the analysis of array properties in assembly code. It identifies address sets that correspond to arrays, represents them using Symbolic Linear Progressions, and uses summarisation to compactly represent global properties on their contents.

In future works, we plan to extend this work to more general data structures, to support the analysis of linked lists for instance. We also plan to improve our prototype, to 1) reduce the number of variables created during the analysis; 2) compare with an implementation based on a numerical domain with lower complexity than Polyhedra (octagons for instance).

Acknowledgments

We would like to thank Giuseppe Lipari for his precious feedback on the paper. We would also like to thank Andrei Florea for chasing some naughty bugs away from our definitions.

Funding

Partially funded by the French National Research Agency (ANR), Corteva project (ANR-17-CE25-0003).

Data availability

The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

1. Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International Conference on Compiler Construction*, 2005.
2. Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 2732–2733. Springer, 2004.
3. Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin Heidelberg, 2010.
4. Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. Static analysis of binary code with memory indirections using polyhedra. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 114–135. Springer, 2019.
5. Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG reconstruction from unstructured programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, 2011.
6. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.
7. Aaron R Bradley, Zohar Manna, and Henny B Sipma. What's decidable about arrays? In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006.
8. David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.
9. Stefan Bygde, Björn Lisper, and Niklas Holsti. Fully bounded polyhedral analysis of integers with wrapping. *Electronic Notes in Theoretical Computer Science*, 288:3–13, 2012.
10. Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4), 2016.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (PLDI'77)*, pages 238–252. ACM, 1977.
12. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, 2011.
13. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, pages 84–96. ACM, 1978.
14. Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. Digging for data structures. In *OSDI*, volume 8, 2008.
15. Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.
16. Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529. Springer, 2004.
17. Denis Gopan and Thomas Reps. Lookahead widening. In *International Conference on Computer Aided Verification*, pages 452–466. Springer, 2006.
18. Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1):338–350, 2005.

19. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
20. Peter Habermehl, Radu Iosif, and Tomáš Vojnar. What else is decidable about integer arrays? In *International Conference on Foundations of Software Science and Computational Structures*, pages 474–489. Springer, 2008.
21. Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, volume 43, pages 339–348, 2008.
22. Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Case studies on invariant generation using a saturation theorem prover. In *Mexican International Conference on Artificial Intelligence*, pages 1–15. Springer, 2011.
23. Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer Aided Design*, 2010.
24. Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.
25. Jiangchao Liu and Xavier Rival. Abstraction of optional numerical values. In *Asian Symposium on Programming Languages and Systems*, pages 146–166. Springer, 2015.
26. Jiangchao Liu and Xavier Rival. An array content static analysis based on non-contiguous partitions. *Computer Languages, Systems & Structures*, 47:104–129, 2017.
27. Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1), 2006.
28. J. D. Monk. *Mathematical Logic. Graduate Texts in Mathematics*, volume 37, chapter Cylindric Algebras. Springer, 1976.
29. Đurica Nikolić and Fausto Spoto. Inferring complete initialization of arrays. *Theoretical Computer Science*, 484, 2013.
30. Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite, 2012. `http://www.cs.ucla.edu/pouchet/software/polybench`.
31. Ganesan Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, 1999.
32. Thomas Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *Compiler Construction*, pages 16–35. Springer, 2008.
33. Rathijit Sen and YN Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pages 39–48. IEEE, 2007.
34. Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In *18th Working Conference on Reverse Engineering (WCRE'11)*. IEEE, 2011.
35. Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.
36. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.
37. Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.
38. Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.