Discovering properties about arrays in simple programs

Nicolas Halbwachs, Mathias Péron

Discovering Properties about Arrays in Simple Programs*

Nicolas Halbwachs, Mathias Péron
Verimag - CNRS†
Grenoble – France
Nicolas.Halbwachs@imag.fr/Mathias.Peron@imag.fr

Abstract

Array bound checking and array dependency analysis (for parallelization) have been widely studied. However, there are much less results about analyzing properties of array *contents*. In this paper, we propose a way of using abstract interpretation for *discovering* properties about array contents in some restricted cases: one-dimensional arrays, traversed by simple "for" loops. The basic idea, borrowed from [15], consists in partitioning arrays into symbolic intervals (e.g., $[1, i-1]$, $[i, i]$, $[i+1, n]$), and in associating with each such interval I and each array A an abstract variable A_I; the new idea is to consider *relational* abstract properties $\psi(A_I, B_I, ...)$ about these abstract variables, and to interpret such a property pointwise on the interval I: $\forall \ell \in I, \psi(A[\ell], B[\ell], ...)$. The abstract semantics of our simple programs according to these abstract properties has been defined and implemented in a prototype tool. The method is able, for instance, to discover that the result of an insertion sort is a sorted array, or that, in an array traversal guarded by a "sentinel", the index stays within the bounds.

1 Introduction

Although array bound checking was a motivation of the very first work on abstract interpretation [9], analyzing properties of array *contents* was considered only recently. The reason is, of course, that the general problem is difficult: array indexing induces complex semantics, and in particular the possibility of aliasing; moreover, since the size of an array can be large or unknown, it represents a large or unbounded number of variables. In this paper, we propose a way of using abstract interpretation for discovering properties about array contents in some restricted cases. First, we restrict ourselves to one-dimensional arrays and "simple programs", which manipulate arrays only by sequential traversal: typically, *for* loops incrementing (or decrementing) their index at each iteration, and accessing arrays by simple expressions (constant translations) of the loop index. Fig. 1 shows several examples of such "simple programs", which will be used throughout the paper. Second, we consider a restricted class of properties: while a quite general kind of property about arrays $A_1, A_2, \ldots, A_m$ could be written

$$\forall \ell \in D, \psi(A_1[f_1(\ell)], \ldots, A_m[f_m(\ell)], x_1, \ldots, x_p) \quad (1)$$

where D is some set of values for indices, ψ is some scalar property about content values, $f_1, \ldots, f_m$ are general index functions, and $x_1, \ldots, x_p$ are scalar variables, we will only consider properties of the form

$$\forall \ell \in I, \psi(A_1[\ell + k_1], \ldots, A_m[\ell + k_m], x_1, \ldots, x_p) \quad (2)$$

where I is an *interval*, and $k_1, \ldots, k_m$ are integer constants.

For instance, our method will discover automatically the following properties:

- at the end of "Array maximum" (Fig. 1.a):

$$\forall \ell \in [1, n], A[\ell] \leq max \quad (3)$$

- at the end of "Array copy" (Fig. 1.b):

$$\forall \ell \in [1, n], A[\ell] = B[\ell] \quad (4)$$

- at the end of "Insertion sort" (Fig. 1.c):

$$\forall \ell \in [2, n], A[\ell] \geq A[\ell - 1] \quad (5)$$

So, in spite of severe restrictions both on programs and properties, our method allows interesting properties to be found about non trivial programs.

Related work: The automatic analysis of properties of array contents was considered only recently. [6, 18] study decidable logics for expressing such properties. If we restrict ourselves to automatic analysis, an important track initiated by [12] concerns verification of programs with arrays using predicate abstraction [21, 20], possibly improved with counter-example guided refinement [3] and Craig interpolants [19]. All these

*This work has been partially supported by the APRON project of the "ACI Sécurité Informatique" of the French Ministry of Research.

†Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG.

```
max := A[1] ;
for i := 2 to n do
    if max < A[i] then
        max := A[i]
```

a. Array maximum

```
for i := 1 to n do
    A[i] := B[i]
```

b. Array copy

```
for i := 2 to n do
    x := A[i]; j := i − 1 ;
    while j ≥ 1 and A[j] > x do
        A[j + 1] := A[j] ; (⋆)
        j := j − 1
    A[j + 1] := x
```

c. Insertion sort

```
x := A[1] ; i := 2 ; j := n ;
while i ≤ j do
    if A[i] < x then
        A|i − 1] := A[i] ;
        i := i + 1
    else
        while j ≥ i and A[j] ≥ x do
            j := j − 1
        if j > i then
            A[i − 1] := A[j]; A[j] := A[i] ;
            i := i + 1 ; j := j − 1
A[i − 1] := x ;
```

d. Find: segmentation phase of the QuickSort

```
A[1] := 7 ;
for i := 2 to n do
    A[i] := A[i − 1] + 1
```

e. Sequence initialization

```
A[n] := x ; i := 1 ;
while A[i] ≠ x do
    i := i + 1
```

f. Sentinel

```
s := n + 1 ;
for i := 1 to n do
    if s = n + 1 and A[i] ≠ 0 then
        s := i
```

g. First not null

Figure 1: Some simple programs

approaches make use of the property to be proved, while we aim at discovering properties.

Concerning automatic program analysis methods based on a abstract interpretation, a general common approach [4, 14] is by *summarizing* a collection S of variables with one auxiliary variable, say s, managed to satisfy the disjunction of properties of variables in the collection: if s satisfies a property ψ so do all the variables $v \in S$.

In [4], this approach is called "*array smashing*": all the cells of an array A are subsumed by one variable a, of the same type as the cells. Initially, a is given the strongest known property satisfied by all the initial values of the cells of A. Each assignment "$A[i] := e$" to an array cell is replaced by a *weak update* of the auxiliary variable: the weak update of an expression e to a variable a (it will be noted $a \sqcup\!\!= e$) can be interpreted as a non deterministic choice between keeping a unchanged and actually performing the assignment (or *strong update*) $a := e$. For instance, if all the cells of an array A are known to satisfy some property ψ, and if the expression e is known to satisfy ψ', then after an assignment $A[i] := e$ all the cells of A are known to satisfy $\psi \sqcup \psi'$ (where $\sqcup$ is the least upper bound operator on properties), which is exactly the effect of a weak update $a \sqcup\!\!= e$, if a is known to satisfy ψ before. The problem with this approach is that the weak assignment can only lose information; moreover tests on individual cells don't bring any information. One needs to know (i.e., the user has to provide) an initial property satisfied by all the array cells, and the analysis can only weaken this initial knowledge. As a consequence, the results are generally very unprecise.

[15, 13] proposes a significant improvement, by partitioning the index domain (say, $[1..n]$) into several symbolic intervals (e.g., $I_1 = [1\,..\,i{-}1]$, $I_2 = [i, i]$, $I_3 = [i{+}1\,..\,n]$), and associating with each subarray $A[I_k]$ a summary auxiliary variable a_k, managed so that

$$\psi(a_k) \Rightarrow \psi(A[\ell]), \forall \ell \in I_k \tag{6}$$

In order to reduce the loss of information due to weak updates, each cell $A[i]$ appearing in the left-hand side of an assignment or in a test constitutes a singleton in the partition, so the assignment can be interpreted as a strong update of the corresponding summary variable. This technique is able to discover our property (3) at the end of the "Array maximum" program. Concerning the "Array copy", if we know that all the cells of B are positive, it can deduce that so are all the cells of A after the copy. So, the method is much more effective in discovering global properties of array contents. However, it cannot discover relations between the contents of different cells. For instance, it is not able to discover that $A[\ell] = B[\ell]$ for all ℓ (property (4)) at the end of "Array copy". The paper proposes a technique to check such properties, which succeeds for our properties (4) and (5), but these candidate invariants must be provided by the user.

While in our property scheme (2), the indices will be quantified over intervals chosen in a fixed partition like in [15, 13], [16] considers more general pointwise properties of the form $\forall \ell, \varphi(\ell) \Rightarrow \psi(A[\ell] \ldots)$. This generality involves difficulties, since in such properties, assumptions φ on indices must be under-approximated. Moreover, the user must provide some template for the properties that should be discovered.

[5] deals with the same kind of problems in a quite different context, which is the analysis of data-sensitive programs manipulating single linked lists. The considered data structure is rather different, and the method is specialized to ordering relations. Moreover, as for Cousot's *parametric abstract domains* [8], the considered properties express relations between *all* the elements of two data collections, while the essence of our approach is to express *pointwise* relations.

Contribution: Our main contribution, in the present paper, is to propose a fully automatic method to *discover relations* between array cells. For that, we use the same partitioning approach as [15, 13], but the auxiliary variables a_k are not summary variables (let us call them *slice variables*), they are interpreted in a much more restrictive sense: we generalize the interpretation (6) by giving the following sense to *relations* between slice variables:

$$\psi(a_k, b_k) \Rightarrow \psi(A[\ell], B[\ell]), \forall \ell \in I_k \tag{7}$$

Moreover, we will introduce *shift variables*, representing fixed translations of array slices, in order to be able to express relations like

$$\psi(A[\ell], A[\ell + k_1], B[\ell + k_2]), \forall \ell \in I$$

as announced at (2).

The paper is organized as follows: in Section 2, we give a better intuition of the method, by dealing informally with the "Array copy" example. Section 3 makes precise the kind of programs we consider, and Section 4 defines the abstract properties we shall deal with. All necessary operations on these properties are defined in Section 5. Section 6 describes our prototype implementation and the experiments performed. We conclude the paper with some perspectives.

2 An Intuitive Example

We first give a very informal intuition of the method. Let us consider the program "Array copy" of Fig. 1.b. As in [15], since there is an assignment to $A[i]$, the set $[1, n]$ of index values is split into three intervals:

$$I_1 = [1, (i-1)] \ , \ I_2 = [i, i] \ , \ I_3 = [(i+1), n]$$

and with each array, we associate three slice variables, say:

$$A \rightarrow (a_1, a_2, a_3) \ \ B \rightarrow (b_1, b_2, b_3)$$

which take their values in the same set as array contents. As said before, a property $\psi(a_k)$ should be understood as

$$\forall \ell \in I_k, \psi(A[\ell]).$$

Remarks:

1. If I_k is empty (e.g., I_1 when $i = 1$), $\psi(a_k)$ is true for all ψ (in particular *false*).
2. Intervals I_k are symbolic; in particular, the emptiness of I_k depends on the value of the index i.

With these auxiliary variables, interpreted in that way:

- the assignment "$A[i] := B[i]$" can be abstracted into "$a_2 := b_2$"
- when i is incremented, its previous value moves from I_2 to I_1, and its current value is extracted from I_3 to become the only element of I_2. So, the index incrementation involves
  - a "weak update" $(a_1, b_1) \sqcup\!= (a_2, b_2)$; the assignment is weak because I_1 is not a singleton.
  - an assignment $(a_2, b_2) := (a_3, b_3)$, which is strong because I_2 is a singleton (This just an intuitive justification. In fact, such a strong assignment would result in the property $a_2 = a_3$, which makes no sense according to formula (7) since it relates variables corresponding to distinct intervals with different size. The actual interpretation is more complex, see §5.5).

In summary, instead of considering the initial program with arrays, we can analyze the following program without arrays:

```
1 i := 1 ;
2 while i ≤ n do
3 |   a2 := b2 ;
4 |   i := i + 1 ;
5 |   (a1, b1) ⊔= (a2, b2) ;
6 |   (a2, b2) := (a3, b3)
```

and interpret, in the results, the properties of (a_k, b_k) as defined by formula (7).

Let us assume that classical analyses are available, which take into account simple inequalities of indices (e.g., difference bound matrices [11] or octagons [23]), and equalities of array contents: so, after the assignment "$a_2 := b_2$", we know that $a_2 = b_2$.

Now, the analysis of our program without arrays provides:

- At the first iteration:
  - After line 2: $i = 1$, which implies $I_1 = \emptyset$, so *false*(a_1, b_1) (cf. remark 2 above).

- after line 3: $i = 1, \mathit{false}(a_1, b_1), (a_2 = b_2)$
- after line 6: $i = 2, (a_1 = b_1), (a_2 = a_3), (b_2 = b_3)$, since $\mathit{false}(a_1, b_1) \sqcup (a_1 = b_1) = (a_1 = b_1)$;

- At the second iteration:
  - at line 2, the property on i is widened to $1 \leq i \leq n+1$, and a least upper bound is taken on other properties, giving $1 \leq i \leq n+1, (a_1 = b_1)$.
  - after line 3: $2 \leq i \leq n, (a_1 = b_1), (a_2 = b_2)$
  - after line 6: $2 \leq i \leq n, (a_1 = b_1), (a_2 = a_3), (b_2 = b_3)$, and the iteration converges;

- So, the final result at the end of the program is $(i = n+1)$, $(a_1 = b_1)$, which can be interpreted as the expected result

$$(i = n+1), \forall \ell \in [1, i-1], A[\ell] = B[\ell].$$

3 Simple Programs

For simplicity, we assume that programs manipulate only data types "integer" (for indices), "content" (an arbitrary type), and "array of content". The following sets will be considered (with associated meta-variables):

- Integer constants: $\mathbb{Z}$ ($\ni k$)
- Integer parameters (or non assignable integer variables, used for parametric array size): *Params* ($\ni n$)
- Integer variables: *Indices* ($\ni i, j$)
- Content constants: $\mathbb{C}$
- Content variables: *CVars* ($\ni x, y$)
- Array variables: *Arrays* ($\ni A, B$)

3.1 Syntax

Fig. 2 gives the abstract syntax of our simple programs. Some of the imposed restrictions are just for simplicity, others are necessary for the analysis to give good results. As usual in abstract interpretation, a more general language can be considered, either by defining specific extensions to the analysis, or by abstracting away the extended features.

We ignore the declarations, so a program is simply a (sequence of) statement(s). Statements can be assignments to content variables or array elements (the syntax of content expressions ⟨*C-exp*⟩ is left unspecified, since it depends on the content type), assignment to integer variables (which must *not* be loop indices), "for" loops (which are restrictions of C-like "for" constructs: in the initialization of the loop index "i := ⟨*Iexp*⟩", the expression ⟨*Iexp*⟩ may not depend on i; the loop progress statement can only be an index incrementation ($++$) or decrementation ($--$); the loop index may not be assigned inside the loop), conditionals, and sequences. The syntax of "for" loops is convenient to express the wanted restrictions, but for detailing the analysis of examples, such loops will often be decomposed into "i := ⟨*Iexp*⟩; while⟨*cond*⟩{⟨*statement*⟩; i⟨*progress*⟩}". Index expressions are restricted to constants or parameters and sums of an index or parameter and a constant. Conditions are conjunctions and/or disjunctions of atomic conditions, whose syntax will depend on the lattices used in the analysis (see §4.1): the conditions on contents are supposed to express at least equalities (e.g., $x = A[i+1]$), those on indices are supposed to express at least potentials (e.g., $i \leq j - 3$).

⟨*program*⟩ ::= ⟨*statement*⟩
⟨*statement*⟩ ::= ⟨*left-part*⟩ := ⟨*C-exp*⟩
| i := ⟨*Iexp*⟩
| for(i := ⟨*Iexp*⟩; ⟨*cond*⟩; ⟨*progress*⟩) ⟨*statement*⟩
| if ⟨*cond*⟩ ⟨*statement*⟩ ⟨*statement*⟩
| ⟨*statement*⟩ ; ⟨*statement*⟩
⟨*left-part*⟩ ::= x | A[⟨*Iexp*⟩]
⟨*Iexp*⟩ ::= k | n | i | ⟨*Iexp*⟩ + k
⟨*cond*⟩ ::= ⟨*Icond*⟩ | ⟨*Ccond*⟩
| ⟨*cond*⟩ and ⟨*cond*⟩ | ⟨*cond*⟩ or ⟨*cond*⟩
⟨*progress*⟩ ::= $++$ | $--$

Figure 2: The syntax of simple programs

3.2 Semantics

Arrays will be indexed from 1 to m, where m is the size of the array. We are concerned with the analysis of array contents, not with array bound checking, which we assume to be solved by other means. As a consequence, we don't want to bother about access out of bounds. This is reflected by considering an array value to be a function $\mathbb{Z} \mapsto \mathbb{C}_\perp$ from all relative integers to the domain of contents completed with a $\perp$ element: of course, an array value A is restricted to return a non $\perp$ value exactly on an interval $[1, m]$.

Let *States* denote the set of states of a program. A state is a triple $(\mathcal{I}, \mathcal{C}, \mathcal{A})$, where $\mathcal{I}$: (*Indices* ∪ *Params*) $\mapsto \mathbb{Z}$ is a valuation for indices and parameters, $\mathcal{C}$: *CVars* $\mapsto \mathbb{C}$ is a valuation for content variables, and $\mathcal{A}$: *Arrays* $\mapsto (\mathbb{Z} \mapsto \mathbb{C}_\perp)$ is a valuation for arrays. The semantics of statements is described in Fig. 3 as functions from *States* to *States*.

$[\![.]\!]$: $\langle \textit{statement} \rangle \mapsto \textit{States} \mapsto \textit{States}$
$\langle \textit{Cexp} \rangle \mapsto \textit{States} \mapsto \mathbb{C}$
$\langle \textit{Iexp} \rangle \mapsto \textit{States} \mapsto \mathbb{Z}$
$\langle \textit{cond} \rangle \mapsto \textit{States} \mapsto \mathbb{B}$

$[\![x := \langle \textit{C-exp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = (\mathcal{I}, \mathcal{C}[[\![\langle \textit{C-exp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A})/x], \mathcal{A})$
$[\![i := \langle \textit{Iexp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = (\mathcal{I}[[\![\langle \textit{Iexp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A})/i], \mathcal{C}, \mathcal{A})$
$[\![A[\langle \textit{Iexp} \rangle] := \langle \textit{C-exp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = (\mathcal{I}, \mathcal{C}, \mathcal{A}[\mathcal{F}/A])$
where $\mathcal{F} = \lambda z. \begin{cases} \mathcal{A}(A)(z) \text{ if } z \neq [\![\langle \textit{Iexp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \\ [\![\langle \textit{C-exp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \text{ otherwise} \end{cases}$
$[\![\text{for(i} := \langle \textit{Iexp} \rangle; \langle \textit{cond} \rangle; \langle \textit{prog} \rangle) \langle \textit{stat} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) =$
$[\![\text{while} \langle \textit{cond} \rangle \langle \textit{stat} \rangle i \langle \textit{prog} \rangle]\!](\mathcal{I}[[\![\langle \textit{Iexp} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A})/i], \mathcal{C}, \mathcal{A})$
$[\![\text{while} \langle \textit{cond} \rangle \langle \textit{stat} \rangle i \langle \textit{prog} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) =$
$\begin{cases} (\mathcal{I}, \mathcal{C}, \mathcal{A}) \text{ if } [\![\langle \textit{cond} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = \text{false} \\ [\![\langle \textit{stat} \rangle; i \langle \textit{prog} \rangle; \text{while} \langle \textit{cond} \rangle \langle \textit{stat} \rangle i \langle \textit{prog} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \text{ otherwise} \end{cases}$
$[\![i{+}{+}]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = (\mathcal{I}[(\mathcal{I}(i) + 1)/i], \mathcal{C}, \mathcal{A})$
$[\![i{-}{-}]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = (\mathcal{I}[(\mathcal{I}(i) - 1)/i], \mathcal{C}, \mathcal{A})$
$[\![\text{if} \langle \textit{cond} \rangle \langle \textit{stat1} \rangle \langle \textit{stat2} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) =$
$\begin{cases} [\![\langle \textit{stat1} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \text{ if } [\![\langle \textit{cond} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = \text{true} \\ [\![\langle \textit{stat2} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \text{ otherwise} \end{cases}$
$[\![\langle \textit{stat1} \rangle; \langle \textit{stat2} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) = [\![\langle \textit{stat2} \rangle]\!]([\![\langle \textit{stat1} \rangle]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}))$

Figure 3: Semantics of simple programs

4 Array Content Properties

4.1 Lattices

Throughout the paper, we assume the existence of two analyses, the former concerning the behavior of indices, and the later concerning contents. In some sense, our method is parameterized by these analyses:

- The analysis of indices is based on a lattice $(L_{\mathbb{Z}}, \sqsubseteq_{\mathbb{Z}}, \sqcap_{\mathbb{Z}}, \sqcup_{\mathbb{Z}}, \top_{\mathbb{Z}}, \bot_{\mathbb{Z}})$ of properties over the set *Indices* of index variables. Elements of $L_{\mathbb{Z}}$ will be noted ϕ. $L_{\mathbb{Z}}$ must be a relational lattice, at least as powerful as potential constraints (i.e., systems of inequalities of the form $i - j \leq k$, $k_1 \leq i \leq k_2$, often implemented as "Difference Bound Matrices" [11, 1]), and defining convex properties. Candidates for $L_{\mathbb{Z}}$ are potential constraints, octagons [23], octahedra [7], or polyhedra [10]. In this paper we will consider $L_{\mathbb{Z}}$ to be the lattice of potential constraints. We will also use an extension $L'_{\mathbb{Z}}$ of $L_{\mathbb{Z}}$, expressing properties over *Indices* $\cup \{\ell\}$, where ℓ is a new special variable (used for quantification). Elements of $L'_{\mathbb{Z}}$ will be noted φ. The same notations will be used for operations in $L_{\mathbb{Z}}$ and $L'_{\mathbb{Z}}$, and $L_{\mathbb{Z}}$ will often be implicitly plunged into $L'_{\mathbb{Z}}$.

- The analysis of contents is based on a lattice $(L_{\mathbb{C}}, \sqsubseteq_{\mathbb{C}}, \sqcap_{\mathbb{C}}, \sqcup_{\mathbb{C}}, \top_{\mathbb{C}}, \bot_{\mathbb{C}})$, of which we only assume that it is able to express equality relations. Elements of $L_{\mathbb{C}}$ will be noted ψ.

In many classical examples, it happens that array contents are numbers. In that case, we can choose $L_{\mathbb{C}} = L_{\mathbb{Z}}$, but this choice is not compulsory.

4.2 Partitions

Following [15], our method relies on the choice of a symbolic partition of the index domain. We formalize such a partition as a finite set $\mathcal{P} = (\varphi_p)_{p \in P}$ of properties in $L'_{\mathbb{Z}}$, such that

$$\bigsqcup_{p \in P}{}_{\mathbb{Z}} \varphi_p = \top_{\mathbb{Z}} \ , \ \forall p, p' \in P, (p \neq p') \Rightarrow \varphi_p \sqcap_{\mathbb{Z}} \varphi_{p'} = \bot_{\mathbb{Z}}$$

Elements φ_p of a partition will be called *slices*.

An example of partition is

$$\begin{array}{ll} \varphi_0 = (i < 1) & \varphi_1 = (\ell < 1 \leq i \leq n) \\ \varphi_2 = (1 \leq \ell < i \leq n) & \varphi_3 = (1 \leq \ell = i \leq n) \\ \varphi_4 = (1 \leq i < \ell \leq n) & \varphi_5 = (1 \leq i \leq n < \ell) \\ \varphi_6 = (n < i) & \end{array}$$

A different partition will be considered for each loop in the program. The choice of the partition, for a given loop, is performed automatically from the text of the program according to the following rules:

(i) the partition of a nested loop should refine the partition of the outer loop;

(ii) for a loop "for(i := Iexp; cond; $++$)", the partition should distinguish the cases ($\ell <$ Iexp) and ($\ell \geq$ Iexp); conversely, for a loop "for(i := Iexp; cond; $--$)", the partition should distinguish the cases ($\ell >$ Iexp) and ($\ell \leq$ Iexp).

(iii) for each "A[Iexp]" appearing either in the left part of an assignment, or in a condition, the partition should distinguish between ($\ell <$ Iexp), ($\ell =$ Iexp), and ($\ell >$ Iexp).

In practice, a partition doesn't have to cover the whole domain of indices: it is enough that it covers all the valuations that are reachable during the program execution; in particular, the cases where ℓ is outside the array bounds are not considered. A preliminary analysis of the indices generally provides a restricted domain for each loop. The left column of Fig. 4 shows a realistic example of partition, where, since it is known that $0 \leq j < i \leq n + 1$, situations like $i \leq 0$ or $j \geq i$ are not considered.

Remarks:

- Although a slice φ_p in a partition is intended to specify an interval for the specific variable ℓ, this interval depends on the valuation $\mathcal{I}$ of indices.

- Moreover, φ_p may involve constraints on $\mathcal{I}$. We will note $\overline{\varphi_p}$ the property $\exists \ell. \varphi_p$ which summarizes the constraints on indices in the slice φ_p.

$\phi = 0 \leq j \leq i-1, 2 \leq i \leq n$

$\varphi_1 = (1 = \ell < j < i)$	$\psi_1 = \top_{\mathbb{C}}$
$\varphi_2 = (1 = j = \ell < i)$	$\psi_2 = \top_{\mathbb{C}}$
$\varphi_3 = (1 = j+1 = \ell < i)$	$\psi_3 = (a^0 > x)$
$\varphi_4 = (2 \leq \ell < j < i)$	$\psi_4 = (a^0 \geq a^{-1})$
$\varphi_5 = (2 \leq j = \ell < i)$	$\psi_5 = (a^0 \geq a^{-1})$
$\varphi_6 = (2 \leq j+1 = \ell < i)$	$\psi_6 = (a^0 > x, a^0 \geq a^{-1})$
$\varphi_7 = (1 \leq j+1 < \ell < i)$	$\psi_7 = (a^0 \geq a^{-1} > x)$
$\varphi_8 = (2 \leq \ell = j+1 = i)$	$\psi_8 = (a^0 = x)$
$\varphi_9 = (1 \leq j+1 < \ell = i)$	$\psi_9 = (a^0 \geq a^{-1} > x)$
$\varphi_{10} = (1 \leq j+1 \leq i, 2 \leq i < \ell)$	$\psi_{10} = \top_{\mathbb{C}}$

Figure 4: Example of abstract value for insertion sort

- As soon as an equation ($\ell =$ Iexp) appears in the definition of a slice, the slice is a *singleton*: for any $\mathcal{I}$ there is at most one ℓ in the slice. Singleton slices play an important role, because their corresponding slice variables may be dealt with as scalar variables. We note *Single*($\mathcal{P}$) the set of singletons in $\mathcal{P}$.

Slice compatibility: Let's recall that $\overline{\varphi_p}$ denotes the constraints on indices involved by a slice φ_p. Now, if $\overline{\varphi_p} \sqcap \overline{\varphi_{p'}} = \bot_{\mathbb{Z}}$, the slices φ_p and $\varphi_{p'}$ induce contradictory constraints, so these slices cannot be both non-empty in a same state. There are said to be *incompatible*. We shall use a notion of compatibility relative to some formula ϕ on indices: two slices φ_p and $\varphi_{p'}$ will be said to be ϕ-*compatible*, if and only if $\phi \sqcap \overline{\varphi_p} \sqcap \overline{\varphi_{p'}} \neq \bot_{\mathbb{Z}}$.

4.3 Abstract Values

As announced in the introduction, our analysis will make use of *slice variables*, possibly *shifted*. The choice of these variables will be explicited later, but we already assume the existence of a finite set $\{a^z\}$ of slice variables: this notation associates uppercase letters (e.g., A, B) for arrays, with lowercase letters (a, b) for slice variables; the exponent z is a relative integer, called the *shift*. For a given valuation $\mathcal{I}$ of index variables and a given slice φ_p, the slice variable a^z represents the subarray $\{A[\ell + z] \mid \varphi_p(\mathcal{I}, \ell)\}$.
Remark: the disappearance of the subscript p, referring to a symbolic slice, on slice variables is *not* a shortcut. As we saw during the analysis of our intuitive example (§2), we are interested in the join of properties linked to different symbolic slices (e.g., weak update). If these properties use the same slice variables, the join operation can be performed in a classical lattice (here $L_{\mathbb{C}}$). In the example, in place of joining $a_1 = b_1$ and $a_2 = b_2$ the join will be between two identical properties $a^0 = b^0$.

Given a partition $(\varphi_p)_{p \in P}$ and a set of slice variables $\{a^z\}$, an *abstract value* Ψ consists of

- a property $\phi \in L_{\mathbb{Z}}$ of index variables
- a tuple $(\psi_p)_{p \in P}$ of properties ($\in L_{\mathbb{C}}$) of slice variables and content variables.

The concretization $\gamma(\Psi)$ of an abstract value Ψ is a set of states ($\gamma(\Psi) \subseteq$ *States*) defined as follows:

$$(\mathcal{I}, \mathcal{C}, \mathcal{A}) \in \gamma(\Psi) \Leftrightarrow \begin{cases} \bullet\ \phi(\mathcal{I}) \\ \bullet\ \forall p \in P, \forall \ell \text{ such that } \varphi_p(\mathcal{I}, \ell) \\ \qquad \psi_p[A[\ell + z]/a^z](\mathcal{C}, \mathcal{A}) \end{cases}$$

(in the formula above, $\psi_p[A[\ell + z]/a^z](\mathcal{C}, \mathcal{A})$ means that $(\mathcal{C}, \mathcal{A})$ satisfy the formula ψ_p where each variable a^z has been replaced by $A[\ell + z]$).

Example: Fig. 4 shows the abstract value associated with the entry of the inner loop of the insertion sort (Fig. 1.c) at the end of the analysis. For instance, (φ_7, ψ_7) expresses that

$$\forall \ell, (1 \leq j+1 < \ell < i) \Rightarrow A[\ell] \geq A[\ell - 1] > x.$$

We note *Vars*(ψ_p) the set of variables appearing in (i.e., constrained by) ψ_p.

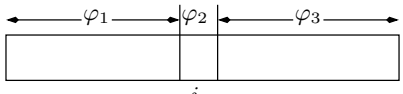
For each partition $\mathcal{P}$, the set of abstract values forms a lattice $(L_{\mathbb{A}}(\mathcal{P}), \sqsubseteq_{\mathbb{A}}, \sqcap_{\mathbb{A}}, \sqcup_{\mathbb{A}}, \top_{\mathbb{A}}, \bot_{\mathbb{A}})$. The lattice operations and all other necessary operations on abstract values are described in the following section.

5 Operations on Abstract Values

5.1 Normalization

An abstract value $\Psi = (\phi, (\varphi_p)_{p \in P})$ is a complex object which must be kept consistent. For instance, if its concretization is empty, it should be normalized to $\bot_{\mathbb{A}}$. There are other needs for normalization, which are taken into account by the following conditional rewriting rules:

1. (empty slice) As said before, when a slice φ_p is empty, the corresponding ψ_p can be any formula, in particular the strongest one $\bot_{\mathbb{C}}$. Hence the rule:
$$(\phi \sqcap_{\mathbb{Z}} \overline{\varphi_p} = \bot_{\mathbb{Z}}) \ \Rightarrow\ \psi_p \rightarrow \bot_{\mathbb{C}}.$$
2. (unsatisfiable index property) If ϕ is not satisfiable, $\gamma(\Psi)$ is empty:
$$\phi = \bot_{\mathbb{Z}} \ \Rightarrow\ \Psi \rightarrow \bot_{\mathbb{A}}.$$
3. (consistency of constraints on scalars) The knowledge about scalar contents variables in each ψ_p should be propagated to all slices ϕ-compatible slices with φ_p. For instance, after the assignment "$x = A[i]$", in the slice $[i, i]$ we know that x has the same properties than $A[\ell]$. The part of this knowledge that does not depend on ℓ



$\phi = (i \geq 1)$

$\overline{\varphi_1} = (i \geq 2)$
$\overline{\varphi_2} = (i \geq 1)$
$\overline{\varphi_3} = (i \geq 1)$

$\psi_1 = (x \geq 6)$
$\psi_2 = (x \geq 5)$
$\psi_3 = (n+5 \geq x \geq 5)$

normalization

$\psi_1 = (n+5 \geq x \geq 6)$
$\psi_2 = (n+5 \geq x \geq 5)$
$\psi_3 = (n+5 \geq x \geq 5)$

Figure 5: Example for consistency of scalars properties

(e.g., $x \geq 0$) must be propagated to slices $[1, i-1]$ and $[i+1, n]$. Let $Sc(\psi_p)$ be the projection of ψ obtained by existential quantification of all slice variables a^z. Then

$$(\overline{\varphi_p} \sqcap_{\mathbb{Z}} \phi) \sqsubseteq_{\mathbb{Z}} \overline{\varphi_{p'}} \;\Rightarrow\; \psi_p \rightarrow \psi_p \sqcap_{\mathbb{C}} Sc(\psi_{p'}).$$

Fig. 5 shows an abstract value on partition $\{\varphi_1 = (1 \leq \ell < i \leq n), \varphi_2 = (1 \leq \ell = i \leq n), \varphi_3 = (1 \leq i < \ell \leq n)\}$, followed by the result of the application of the rule. Such an abstract value will appear for instance, in the program

$$i := 1; x := 5; \textbf{ for } i := 1 \textit{ to } n \textbf{ do } x := x + 1$$

at the head of the loop.

4. (consistency of shifts) If some z-shift of a slice φ_p is included inside another slice $\varphi_{p'}$, the constraints on a^{-z} in $\varphi_{p'}$ should apply to a^0 in φ_p. For instance, if we know that $\forall \ell \in [2, n], A[\ell] \geq A[\ell - 1]$, we may conclude that $\forall \ell \in [1, 1], A[\ell] \leq A[\ell+1]$; that is, for $\varphi_1 = (\ell = 1)$ and $\varphi_2 = (2 \leq \ell \leq n)$, if ψ_2 implies $(a^{-1} \leq a^0)$ then φ_1 should imply $(a^0 \leq a^1)$. Let us note $\varphi_p \oplus z = \varphi_p[\ell - z/\ell]$ and $\psi_{p'} \boxplus z = \psi_{p'}[a^{y-z}/a^y]$. Then the first rule for consistency of shift is

$$\varphi_p \oplus z \sqsubseteq_{\mathbb{Z}} \varphi_{p'} \;\Rightarrow\; \psi_p \rightarrow (\psi_p \sqcap_{\mathbb{C}} (\psi_{p'} \boxplus z)).$$

In the example above, $\varphi_1 \oplus 1 = (\ell = 2) \sqsubseteq_{\mathbb{Z}} \varphi_2$, so ψ_2 must be strengthened with $(a^{-1} \leq a^0) \boxplus -1 = (a^0 \leq a^1)$, as desired.

There is a second rule when the reverse inclusion also holds:

$$\varphi_{p'} \sqsubseteq_{\mathbb{Z}} \varphi_p \oplus z \;\Rightarrow\; \psi_{p'} \rightarrow (\psi_{p'} \sqcap_{\mathbb{C}} (\psi_p \boxplus -z)).$$

For instance, this rule will be applied in the normalization phase of Fig. 8, where

$$\varphi_1 = (\ell = i) \quad \varphi_2 = (\ell = i + 3)$$
$$\psi_1 = (a^0 = x) \quad \psi_2 = (a^0 = a^{-3})$$

we have $\varphi_2 \oplus -3 = \varphi_1$, so both ψ_1 and ψ_2 are rewritten:

$$\psi_1 \rightarrow (a^0 = x, a^3 = a^0)\ ,\ \psi_2 \rightarrow (a^0 = a^{-3}, a^{-3} = x).$$

To make the rules readable, we discarded the use of ϕ. However, better results are obtained if the slices appearing in premises of the rules are intersected with ϕ. For instance, consider the partition of Fig. 5 with $\phi = (i = 2)$. Then we have $(\varphi_1 \sqcap_{\mathbb{Z}} \phi) \sqsubseteq_{\mathbb{Z}} ((\varphi_2 \oplus -1) \sqcap_{\mathbb{Z}} \phi)$ and ψ_1 could be strengthened.

Of course, these rules influence each other. By chance they can be applied step by step, in reverse order w.r.t. the list above (i.e., apply thoroughly the rule (4), then apply thoroughly the rule (3), etc.). Henceforth, we will assume that all abstract values are normalized.

Proposition. Normalization does not change the concretization of formulas: $\Psi \rightarrow \Psi' \;\Rightarrow\; \gamma(\Psi) = \gamma(\Psi')$

5.2 Lattice Operations

After normalization, the lattice operations are straightforward. They are defined on abstract values based on the same partition:

$$(\phi, (\varphi_p)_{p \in P}) \sqcap_{\mathbb{A}} (\phi', (\varphi'_p)_{p \in P}) \;=\; (\phi \sqcap_{\mathbb{Z}} \phi', (\varphi_p \sqcap_{\mathbb{C}} \varphi'_p)_{p \in P})$$

$$(\phi, (\varphi_p)_{p \in P}) \sqcup_{\mathbb{A}} (\phi', (\varphi'_p)_{p \in P}) \;=\; (\phi \sqcup_{\mathbb{Z}} \phi', (\varphi_p \sqcup_{\mathbb{C}} \varphi'_p)_{p \in P})$$

$$\Psi \sqsubseteq_{\mathbb{A}} \Psi' \iff \phi \sqsubseteq_{\mathbb{Z}} \phi' \wedge \forall p \in P, \psi_p \sqsubseteq_{\mathbb{C}} \psi'_p$$

5.3 Change of Partition

When entering a loop, a new index i is introduced, and the partition is refined according to this index. Conversely, when a loop is exited, its index is forgotten, and the partition is simplified accordingly. We need operations for transforming an abstract value Ψ on a partition $\mathcal{P}$ to an abstract value $\Psi' = [\mathcal{P} \rightarrow \mathcal{P}'](\Psi)$ on a partition $\mathcal{P}'$ that refines $\mathcal{P}$, and the converse operation, noted $[\mathcal{P}' \xrightarrow{i} \mathcal{P}]$ which also forgets about the index i. If $\mathcal{P}' = (\varphi'_p)_{p \in P'}$ is a refinement of $\mathcal{P} = (\varphi_p)_{p \in P}$, then, for every $p \in P'$, there exists $f(p) \in P$ such that $\varphi'_p \sqsubseteq_{\mathbb{Z}} \varphi_{f(p)}$. Then,

- for $\Psi = (\phi, (\psi_p)_{p \in P}) \in L_{\mathbb{A}}$, we define

$$[\mathcal{P} \rightarrow \mathcal{P}'](\Psi) = (\phi, (\psi_{f(p)})_{p \in P'})$$

- conversely, for $\Psi' = (\phi', (\psi'_p)_{p \in P'})$ we define

$$[\mathcal{P} \xrightarrow{i} \mathcal{P}'](\Psi') = (\exists i.\phi', (\bigsqcup_{p = f(p')} \psi_{p'})_{p \in P})$$

5.4 Slice Property

Given a symbolic interval $\varphi \in L'_{\mathbb{Z}}$ and an abstract formula $\Psi \in L_{\mathbb{A}}(\mathcal{P})$, we want to extract the strongest property $\psi^{\Psi}(\varphi)$ implied by Ψ on φ. Of course, if there exists $p \in P$ such that $\varphi = \varphi_p$, $\psi^{\Psi}(\varphi)$ is simply ψ_p. However, the operation

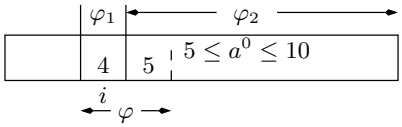


Figure 6: Example for slice property

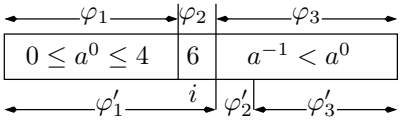


Figure 7: Example for index progression

is more tricky and interesting when φ intersects several slices in $\mathcal{P}$. The first idea is to take the least upper bound of the properties of these slices. However we shall see that additional information can be gained from shifted variables.
We define:

$$\begin{aligned} \mathit{Inter}^\Psi(\varphi) &= \{p \in P \mid \varphi_p \sqcap_\mathbb{Z} \varphi \sqcap_\mathbb{Z} \phi \neq \bot_\mathbb{Z}\} \\ \mathit{Trans}^\Psi(\varphi) &= \{(p,z) \mid a^z \in \mathit{Vars}(\psi_p) \\ &\qquad \text{and } \varphi_p \oplus z \sqcap_\mathbb{Z} \varphi \sqcap_\mathbb{Z} \phi \neq \bot_\mathbb{Z}\} \end{aligned}$$

Then we define

$$\psi^\Psi(\varphi) = \bigsqcup_{p \in \mathit{Inter}^\Psi(\varphi)} \psi_p \sqcap \bigsqcap_{\substack{(q,z) \in \mathit{Trans}^\Psi(\varphi) \\ \varphi_p \sqcap \varphi \sqcap \phi \sqsubseteq \varphi_q \oplus z \sqcap \phi}} \psi_q \boxplus -z$$

We explain this complex formula by means of a small example (see Fig. 6): the partition is $\{\varphi_1 = (i = \ell), \varphi_2 = (i < \ell)\}$, the abstract formula $\Psi = (1 \le i, \{\psi_1 = (a^0 = 4,\ a^1 = 5), \psi_2 = (5 \le a^0 \le 10)\})$. For $\varphi = (i \le \ell \le i+1)$, we have $\mathit{Inter}^\Psi(\varphi) = \{1,2\}$ and $\mathit{Trans}^\Psi(\varphi) = \{(1,1)\}$. We get $\bigsqcup_{p \in \mathit{Inter}^\Psi(\varphi)} \psi_p = (4 \le a^0 \le 10)$. If we take also into account the translated slices intersected by φ (we have $(\varphi \sqcap_\mathbb{Z} \varphi_2) \sqsubseteq_\mathbb{Z} \varphi_1 \oplus 1$) we get the more precise result $\psi = (4 \le a^0 \le 5)$.

5.5 Index Progression

Let us consider index incrementing (the decrementing being symmetrical). When an index i is incremented, the subarray represented by a slice φ_p involving i changes. Let us note $\varphi'_p = \varphi_p[i+1/i]$. Thanks to the operation defined in the previous subsection, we can compute the property $\psi'_p = \psi^\Psi(\varphi'_p)$, which is the property to be satisfied within φ_p after the assignment. So, we can define

$$[i{+}{+}]_\mathbb{A}(\Psi) = \left([i{+}{+}]_\mathbb{Z}(\phi), \left(\psi^\Psi(\varphi_p[i+1/i])\right)_{p \in P}\right).$$

For instance, Fig. 7 represents a property with partition $\{\varphi_1 = (1 \le \ell < i \le n), \varphi_2 = (1 \le \ell = i \le n), \varphi_3 = (1 \le i < \ell \le n)\}$, and $\phi = (1 \le i \le n), \psi_1 = (0 \le a^0 \le 4), \psi_2 = (a^0 = 6), \psi_3 = (a^0 > a^{-1})$. Then, $\varphi'_1 = (1 \le \ell < i{+}1 \le n), \varphi'_2 = (1 \le \ell = i+1 \le n), \varphi'_3 = (1 \le i+1 < \ell \le n)$. We get $[i{+}{+}]_\mathbb{A}(\Psi) = (\phi = (2 \le i \le n+1), \psi_1 = \psi^\Psi(\varphi'_1) = (0 \le a^0 \le 6), \psi_2 = \psi^\Psi(\varphi'_2) = (a^0 > a^{-1}), \psi_3 = \psi^\Psi(\varphi'_3) = (a^0 > a^{-1})$.

5.6 Loop Index Initialization

Index initialization "$i := \mathit{Iexp}$" only occurs when entering a loop. So, it is applied just after the refinement of the partition (cf. §5.3), and nothing is known about i before the assignment. So the transformation is just

$$[i := \mathit{Iexp}]_\mathbb{A}(\Psi) = (\phi \sqcap (i = \mathit{Iexp}), (\psi_p)_{p \in P}).$$

5.7 Content Assignment

A content assignment of an expression *exp*, affects several slice formulas, in which the modified slice variable will be different, the update will be applied strongly or weakly, and *exp* will be translated in terms of slice variables in accordance. First, we address left part issues:

Left part:

- In case of a scalar assignment $x := \mathit{exp}$, the variable x must be strongly updated in all ψ_p.
- In case of an array cell assignment $A[i+k] := \mathit{exp}$, we know that there are singleton slices φ_s where $(\ell = i{+}k)$. In the corresponding ψ_s, a^0 must be strongly assigned. We call Strong_1 the set of such s. Moreover, for each $s \in \mathit{Strong}_1$, there can be also other slices φ_p, ϕ-compatible with φ_s, such that there is some $a^z \in \mathit{Vars}(\psi_p)$ and $\varphi_p \oplus z$ intersects (in fact, contains) φ_s. We call $\mathit{Affect}(\varphi_s)$ the set of such p. If $p \in \mathit{Affect}(\varphi_s)$, ψ_p refers to a shift variable which may be aliased with the assigned variable: if φ_p is a singleton, a^z must be strongly assigned, otherwise it must be weakly assigned in ψ_p. So, we define

$$\begin{aligned} \mathit{Strong}_1 &= \{s \in P \mid \varphi_s \sqsubseteq_\mathbb{Z} (\ell = i + k)\} \\ \mathit{Affect}(\varphi_s) &= \{(p,z) \mid a^z \in \mathit{Vars}(\psi_p), \varphi_s \sqsubseteq_\mathbb{Z} \varphi_p \oplus z\} \\ \mathit{Strong} &= \{s' \in \mathit{Single}(\mathcal{P}) \mid \exists s \in \mathit{Strong}_1, \exists z \in \mathbb{Z}, \\ &\qquad (s', z) \in \mathit{Affect}(\varphi_s)\} \\ \mathit{Weak} &= \{p \in P \setminus \mathit{Single}(\mathcal{P}) \mid \exists s \in \mathit{Strong}_1, \\ &\qquad \exists z \in \mathbb{Z}, (p,z) \in \mathit{Affect}(\varphi_s)\} \end{aligned}$$

Scalar right part: When the right part *exp* does not contain any array access, the postcondition of the assignment is easy to compute. Let us note ψ'_p the slice formulas of $[\mathit{Left} = \mathit{exp}]_\mathbb{A}(\Psi)$. If the left-hand side is scalar, then $\forall p, \psi'_p =$

$[x := exp]_{\mathbb{C}}(\psi_p)$. Otherwise, with the notations defined above:

$\forall s \in \mathit{Strong}_1, \psi'_s = [a^0 := exp]_{\mathbb{C}}(\psi_s)$
$\forall s \in \mathit{Strong}, (s,z) \in \mathit{Affect}(\varphi_{s'}), \psi'_s = [a^z := exp]_{\mathbb{C}}(\psi_s)$
$\forall p \in \mathit{Weak}, (p,z) \in \mathit{Affect}(\varphi_{s'}), \psi'_p = \psi_p \sqcup_{\mathbb{C}} [a^z := exp]_{\mathbb{C}}(\psi_p)$

Right part with array accesses: Without loss of generality assume the right part of the assignment is an expression "$B[j+k]$" (more complex expressions may be reduced to this case using auxiliary scalar variables and successive assignments). If the left-hand side is a scalar variable x, its new value must be computed for each slice. If it is an array cell, we have to perform weak or strong assignments in some slice property ψ_p to some slice variable a^z, p belonging either to Strong_1, or to *Strong*, or to *Weak*. So, we want to get the strongest information about $B[j+k]$ available in some slice φ_p. Two cases occur:

- either $\varphi_p(\ell)$ implies $j+k = \ell+w$, for some w, in which case, $B[j+k]$ is represented by a shift variable b^w: if $\varphi_p \sqcap_{\mathbb{Z}} (\ell' = j+k)$ implies $\ell' = \ell + w$ for some w, then

$$[A[i] := B[j+k]]_{\mathbb{A}}(\psi_p) = [a^z := b^w]_{\mathbb{C}}(\psi_p)$$

- otherwise, we extract the scalar information about $B[j+k]$ using our "slice property" operation (§5.4), as follows: the formula $\psi = \psi^{\Psi}((\ell = j+k) \sqcap_{\mathbb{Z}} \phi \sqcap_{\mathbb{Z}} \overline{\varphi_p})$ gives for b^0 the best information about $B[j+k]$ for the slice φ_p ($(\ell = j+k)$ is ϕ-compatible with φ_p), so $\psi' = \mathit{Sc}(\psi[y/b^0])$, where y is a fresh scalar variable, gives the wanted information for y. Then the result is obtained in a similar way as for a scalar right part reduced to y:

$$[x := B[j+k]]_{\mathbb{A}}(\psi_p) = \exists y.[x := y]_{\mathbb{C}}(\psi_p \sqcap_{\mathbb{C}} \psi')$$

Let $\varphi' = (\ell = j+k)$; For $s \in \mathit{Strong}_1$,

$$[A[i] := B[j+k]]_{\mathbb{A}}(\psi_s) = \exists y.[a^0 := y]_{\mathbb{C}}(\psi_s \sqcap_{\mathbb{C}} \psi')$$

where $\psi' = \mathit{Sc}(\psi^{\Psi}(\varphi' \sqcap_{\mathbb{Z}} \phi \sqcap_{\mathbb{Z}} \overline{\varphi_s})[y/b^0])$.

For $s \in \mathit{Strong}, (s,z) \in \mathit{Affect}(\varphi_{s'})$,

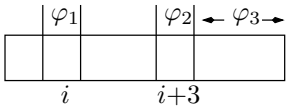
$$[A[i] := B[j+k]]_{\mathbb{A}}(\psi_s) = \exists y.[a^z := y]_{\mathbb{C}}(\psi_s \sqcap_{\mathbb{C}} \psi')$$

where $\psi' = \mathit{Sc}(\psi^{\Psi}(\varphi' \sqcap_{\mathbb{Z}} \phi \sqcap_{\mathbb{Z}} \overline{\varphi_s})[y/b^0])$.

For $p \in \mathit{Weak}, (p,z) \in \mathit{Affect}(\varphi_{s'})$,

$$[A[i] := B[j+k]]_{\mathbb{A}}(\psi_p) = \psi_p \sqcup_{\mathbb{C}} [a^z := y]_{\mathbb{C}}(\psi_p \sqcap_{\mathbb{C}} \psi')$$

where $\psi' = \mathit{Sc}(\psi^{\Psi}(\varphi' \sqcap_{\mathbb{Z}} \phi \sqcap_{\mathbb{Z}} \overline{\varphi_p})[y/b^0])$.



$\psi_1 = (a^0 = x)$
$\psi_2 = (a^0 > x)$
$\psi_3 = (a^0 > x, a^{-1} > x, a^0 \geq a^{-1})$

content assignment $A[i+3] := A[i]$

$\psi_1 = (a^0 = x)$
$\psi_2 = (a^0 = a^{-3})$
$\psi_3 = (a^0 > x, a^{-1} \geq x, a^0 \geq a^{-1})$

normalization

$\psi_1 = (a^0 = x, a^0 = a^3, a^3 = x)$
$\psi_2 = (a^{-3} = x, a^0 = x, a^0 = a^{-3})$
$\psi_3 = (a^0 > x, a^{-1} \geq x, a^0 \geq a^{-1})$

Figure 8: Example for content assignment

Example: On the example shown in Fig 8, we perform the assignment $A[i+3] := A[i]$. We have $s = 2$, $\mathit{Affect}(\varphi_s) = \{(2,0),(3,-1)\}$, $\mathit{Strong}_1 = \{2\}$, $\mathit{Strong} = \{2\}$, $\mathit{Weak} = \{3\}$. So, $\psi'_2 = \psi_2[a^0 := a^{-3}]_{\mathbb{C}}(\psi_2) = (a^0 = a^{-3})$. To compute ψ_3, we have first to compute $\psi' = \mathit{Sc}(\psi^{\Psi}((\ell = i) \sqcap_{\mathbb{Z}} \phi \sqcap_{\mathbb{Z}} \overline{\varphi_3})[y/a^0]) = (a^0 = x)[y/a^0] = (y = x)$, and then $\psi'_3 = \psi_3 \sqcup_{\mathbb{C}} [a^{-1} := y]_{\mathbb{C}}(\psi_3 \sqcap_{\mathbb{C}} \psi') = \psi_3 \sqcup_{\mathbb{C}} [a^{-1} := y]_{\mathbb{C}}(a^0 > x, a^{-1} > x, a^0 \geq a^{-1}, y = x, a^0 > y, a^{-1} > y) = \psi_3 \sqcup_{\mathbb{C}} (a^0 > x, y = x, a^0 > y, a^{-1} = y, a^0 > a^{-1}, a^{-1} = x) = (a^0 > x, a^{-1} \geq x, a^0 \geq a^{-1})$. The rest of the work is done by the normalization. Notice that, in spite of the weak assignment, the property that the slice φ_3 is sorted is preserved.

5.8 Conditional statements

Let us note $\mathit{Cond}(\mathrm{c}, \Psi)$ the strengthening of a formula $\Psi = (\phi, (\psi_p)_{p \in P})$ by the knowledge that the condition "c" is true. According to the syntax of our simple programs, a condition is either a formula on indices, or a formula on contents (including array cells) or a conjunction or a disjunction of conditions. We assume that an elementary condition only on indices is abstracted by a formula $\phi_c \in L_{\mathbb{Z}}$. Then, $\mathit{Cond}(\phi_c, \Psi) = (\phi \sqcap_{\mathbb{Z}} \phi_c, (\psi_p)_{p \in P})$. For elementary conditions only on contents, let us note $\mathit{Cond}(c, \Psi) = (\phi, (\mathit{Cond}(c, \psi_p)_{p \in P}))$. If c does not involve any array expression, it is supposed to be a formula $\psi_c \in L_{\mathbb{C}}$ on content variables, and $\mathit{Cond}(c, \psi_p) = \psi_p \sqcap_{\mathbb{C}} \psi_c$ for all p. Otherwise, if $A[i+k]$ appears in c, there are singleton slices $\varphi_s \sqsubseteq_{\mathbb{Z}} (\ell = i+k)$, for which a represents $A[i+k]$ in ψ_s. As for the assignment, either other array expressions in c can be expressed as translations of $A[i+k]$, and replaced by some a^z in c, or their properties can be extracted using "slice property" operation and reported in c. Anyway, c is transformed into a formula $\psi^s_c \in L_{\mathbb{C}}$. Then, for slices s such that $\varphi_s \sqsubseteq_{\mathbb{Z}} (\ell = i+k)$, $\mathit{Cond}(c, \psi_s) = \psi_s \sqcap_{\mathbb{C}} \psi^s_c$.

Finally, for conditions mixing conditions on indices and on contents, we use the classical lattice operators:

$$\mathit{Cond}(c \text{ or } c', \Psi) = \mathit{Cond}(c, \Psi) \sqcup_{\mathbb{A}} \mathit{Cond}(c', \Psi)$$
$$\mathit{Cond}(c \text{ and } c', \Psi) = \mathit{Cond}(c, \Psi) \sqcap_{\mathbb{A}} \mathit{Cond}(c', \Psi)$$

5.9 Soundness of Operations

Proposition. All the operations defined so far are sound, i.e., $\forall \Psi, \forall (\mathcal{I}, \mathcal{C}, \mathcal{A}) \in \gamma(\Psi)$,

- $\varphi(\mathcal{I}, \ell) \Rightarrow \psi^{\Psi}(\varphi)[A[\ell + z]/a^z](\mathcal{C}, \mathcal{A})$
- $[\![i{+}{+}]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \in [i{+}{+}]_{\mathbb{A}}(\Psi)$
- $[\![i := \mathit{Iexp}]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \in [i := \mathit{Iexp}]_{\mathbb{A}}(\Psi)$
- $[\![x := \mathit{exp}]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \in [x := \mathit{exp}]_{\mathbb{A}}(\Psi)$
- $[\![A[i + k] := \mathit{exp}]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \in [A[i + k] := \mathit{exp}]_{\mathbb{A}}(\Psi)$
- if $[\![c]\!](\mathcal{I}, \mathcal{C}, \mathcal{A}) \in \gamma(\Psi) = \mathit{true}$, then $(\mathcal{I}, \mathcal{C}, \mathcal{A}) \in \mathit{Cond}(c, \Psi)$.

5.10 Widening

We assume that a widening $\nabla_{\mathbb{Z}}$ is available in $L_{\mathbb{Z}}$. For $L_{\mathbb{C}}$ we note $\nabla_{\mathbb{C}}$ either the widening in $L_{\mathbb{C}}$ if any, or the least upper bound $\sqcup_{\mathbb{C}}$ if $L_{\mathbb{C}}$ is of finite depth. Then, a natural definition for a widening operator in $L_{\mathbb{A}}$ is

$$(\phi, (\psi_p)_{p \in P}) \nabla \left(\phi', (\psi'_p)_{p \in P}\right) = \left(\phi \nabla_{\mathbb{Z}} \phi', (\psi_p \nabla_{\mathbb{C}} \psi'_p)_{p \in P}\right).$$

Now, this operator is not completely satisfactory for the following reason: when it happens that some slice φ_p is empty in Ψ (i.e., $\phi \sqcap_{\mathbb{Z}} \varphi_p = \bot_{\mathbb{Z}}$) and not empty in Ψ', the corresponding slice property is widened from $\bot_{\mathbb{C}}$ in the result (since $\psi_p = \bot_{\mathbb{C}}$). In that case, it is better to wait for two meaningful iterates before performing the global widening. So, we define $(\phi, (\psi_p)_{p \in P}) \nabla_{\mathbb{A}} \left(\phi', (\psi'_p)_{p \in P}\right)$ as

$$\begin{cases} \left(\phi', (\psi'_p)_{p \in P}\right) \text{ if } & \exists p, \phi \sqcap_{\mathbb{Z}} \varphi_p = \bot_{\mathbb{Z}} \\ & \text{and } \phi' \sqcap_{\mathbb{Z}} \varphi_p \neq \bot_{\mathbb{Z}} \\ \left(\phi \nabla_{\mathbb{Z}} \phi', (\psi_p \nabla_{\mathbb{C}} \psi'_p)_{p \in P}\right) \text{ otherwise} \end{cases}$$

Proposition. $\nabla_{\mathbb{A}}$ is a widening (i.e., $\Psi \sqcup_{\mathbb{A}} \Psi' \sqsubseteq_{\mathbb{A}} \Psi \nabla_{\mathbb{A}} \Psi'$, and $\nabla_{\mathbb{A}}$ satisfies the chain condition).

5.11 About Shift Variables

The analysis introduces slice and shift variables at different steps: slice variables are created at the creation of new partitions (first entry in a loop); shift variables are introduced during normalization (consistency of shifts) and assignment (array expression in right part). It is important to notice that, even if these variables are created during the analysis, their number remains bounded: they can only appear during the first traversal of a loop, the next iterations can only remove variables.

6 Implementation and Experiments

The method has been implemented in OCaml from a generic analyzer due to B. Jeannet (see `http://bjeannet.gforge.inria.fr/fixpoint/`). When invoking the prototype, one can choose the lattice for indices (intervals or potentials) or a domain for arrays which is a functor on the abstract domains ($L_{\mathbb{Z}}$ and $L_{\mathbb{C}}$) assuming a fixed set of functions on these domains. Of course, classical operations on lattices must be available, but also normalization functions, postcondition of assignment and condition, support (set of involved variables), and support extension/reduction. The program to be analyzed is given as a control-flow graph, in a language inspired from the input language of FAST [2].

The array content analysis takes place after some preliminary steps:

1. First, an index analysis is performed, which checks the array accesses w.r.t. array bounds, and computes invariants about index values at each control point of the program.
2. Second, an analysis of live variables is performed to determine the live indexes in each point
3. Finally, a traversal of strongly connected subcomponents (SCSCs) builds the partition attached with each SCSC, according to the strategy described in §4.2.

The results of these three analyses are merged to get, at each control point, the needed actions for changing the partition (cf. §5.3). The results of analysis (1) are ϕ formulas that are used for simplifying the partition. Only live variables are considered for partitioning.

Then the analysis described in this paper is performed. Notice that for examples 1.f and 1.g, which use disequations over array cells in their conditions, we used the abstract domain of *d*DBM [24] as the lattice of properties over contents ($L_{\mathbb{C}}$). This domain is an extension of the DBM one, handling disequations between pairs of variables.

We give now the results of this analysis on some examples: Insertion sort (Fig. 1.c), a version of the famous "Find" program (Fig. 1.d) used for segmenting arrays in QuickSort, a simple initialization sequence (Fig. 1.e) showing properties mixing indices and contents when the same lattice is used (numerical arrays), a version of the "Sentinel" program (Fig. 1.f), which is a well-known challenge for array bound checking, and a program involving an index assignment: "First not null", extracted from [12]. Of course, simpler programs like "Array maximum" and "Array copy" (Fig 1) give the announced results, which we don't detail here.

	# vert. × # edg.	*# φ_p*	*# shifts avg (max)*	*# iter.*	*time (s)*	*# norm.*	*time (%)*	*norm. avg. time (s) / (#φ_p + #shifts)*
array copy	3 × 3	3	0 (0)	5	0.02	30	-	-
seq. init.	3 × 3	4	0.8 (2)	5	0.05	32	54	0.00016
max. search	5 × 6	4	0.8 (2)	5	0.10	58	50	0.00017
sentinel	3 × 3	9	0 (1)	5	0.21	28	22	0.00019
first n. null	6 × 8	13	0 (1)	6	2.25	89	50	0.00102
insert. sort	9 × 11	4-10	4.6 (11)	7	5.38	169	85	0.00153
find	9 × 13	14	6.7 (14)	6	22.87	171	74	0.00453

Figure 9: Performance results

Results for "Find": Fig. 1.d shows a version of the famous "Find" program [17], used for segmenting an array according to its first element, and used at each step of the "QuickSort". The final results attached by our analyzer to the very end of the program are exactly those expected (i.e., the array is segmented):

- $\phi = (2 \leq i \leq n+1, j = i-1)$
- either $n = 1$, then $i = 2$ and $A[1] = x$
- or $n \geq 2$ and $A[i-1] = x$ and $\forall \ell, 1 \leq \ell < i-1 \Rightarrow A[\ell] < x$ and $\forall \ell, i \leq \ell \leq n \Rightarrow x \leq A[\ell]$

Results for "Insertion sort": The result of the analysis at the end of the program is as simple as the partition at that point: $\forall \ell, 2 \leq \ell \leq n \Rightarrow A[\ell-1] \leq A[\ell]$.

A more interesting result is after the array assignment in the nested loop ($(\star)$ point in Fig.1.c). The situation is: sorting is not terminated ($i \leq n$) and current value in cell j is greater than the key x, so we assign its value to cell $j+1$. We have:

- $\phi = (1 \leq j \leq i-1 \leq n-1)$
- either $j = 1$ then $A[1] = A[2] > x$
  - moreover if $i \geq 3$ then $\forall \ell, 3 \leq \ell \leq i, A[\ell] \geq A[\ell-1] > x$
- or $j \geq 2$, then $\forall \ell, 2 \leq \ell < j, A[\ell] \geq A[\ell-1]$ and $A[j] > x$ and $A[j+1] = A[j] \geq A[j-1]$
  - moreover if $i > j+1$ then $\forall \ell, j+2 \leq \ell \leq i, A[\ell] \geq A[\ell-1] > x$

Results for "Sequence init.": At the end of the program, we get the expected bound on array contents ($\leq n+6$):

- $\phi = (1 \leq n = i-1)$
- $n \geq 1 \Rightarrow A[1] = 7 \leq n+6$
- moreover if $n \geq 2$ then $\forall \ell, 2 \leq \ell \leq n \Rightarrow 8 \leq A[\ell] \leq n+6, 7 \leq A[\ell-1] \leq n+5, A[\ell] = A[\ell-1]+1$

Results for "Sentinel": A more surprising success is the discovery, in the program of Fig. 1.f, that the index cannot exceed the bound n: the "sentinel" x is either found before or at n. At the end of the program we get:

- $\phi = 1 \leq i \leq n$
- $A[i] = x$ and $\forall \ell, 1 \leq \ell < i \Rightarrow A[\ell] \neq x$

It is an interesting case where a property on contents involves a property on indices.

Results for "First not null": The program of Fig. 1.g, was given in [12]. The standard method gives poor results. However, if we enrich the partition by distinguishing the singleton ($\ell = s$) we get the expected property at then end of the program:

- $\phi = (1 \leq s \leq i = n+1)$
- $s \leq n+1 \Rightarrow (\forall \ell, 1 \leq \ell \leq s-1 \Rightarrow A[\ell] = 0)$
- moreover if $s \leq n$ then $A[s] \neq 0$

Performances: Experiments were driven on a Core2 Duo 1.6 GHz, with 2Mo of RAM. In Fig. 9, the first part of the table shows, for each example, the size of the control-flow graph (number of vertices and edges), the size of the partition, the (average and maximum) number of shift variables presents in abstract values during the analysis, the number of iterations before convergence, and the total computation time.

If small examples take less than a half of second to be analyzed, analysis time sharply increases when more slices and shift variables are required. In order to confirm this point, the second part of the table shows the number of normalizations performed during the analysis. This number is directly linked to the number of abstract operations done, which is a good measure of the complexity of the program under analysis. But it does not explain the hight time of the analysis of "Find". The table indicates also the amount of time spent doing normalizations, which most often need more than half the total analysis time. Finally, it gives the average of normalization time divided by the sum of the size of the partition and

the number of shift variables. As expected, the cost to maintain the coherence between slices is dominating, and seems to behave exponentially with respect to the number of slices (compare "Sentinel" and "First not null"), and the number of shift variables (compare "Find" and "First not null").

We know there is room for improvements, particularly focusing on the design of a clever data structure for the partition.

7 Conclusion and Future Work

We have presented a method for automatically discovering properties involving array contents in simple programs. The key idea is to synthesize pointwise relations between array segments. The method is able to deal with problems which are well-known to be difficult. A prototype tool has been implemented and applied to some classical examples, most of which, to our knowledge, were not managed by previously existing methods.

This work deserves to be extended in several directions. Our first task will be to improve our prototype implementation, both concerning the performances and the flexibility; in particular, it should be made fully compatible with the APRON interface (see `http://apron.cri.ensmp.fr/library/`), which would allow all compatible lattices to be used. A more clever choice of the partition would reduce the number of slices and variables. An improved widening is also likely to reduce the number of iterations.

Of course, more general programs must be considered: on one hand, non convex slices should be dealt with to take into account more than simple index incrementation/decrementation; on the other hand, multi-dimensional arrays should be considered as well. We could make use of non-convex partitioning of multi-dimensional arrays as proposed in [22]. Also, we are not far from analyzing programs like QuickSort, but our current prototype does not deal with recursive programs.

Concerning our examples, notice that generally we did not fully verify programs: for "array maximum", we prove that the result is greater than all the array elements, it remains to prove that it belongs to the set of elements; for "insertion sort", we prove that the result is sorted, it remains to prove that it is a permutation of the initial array, and similarly for "find". So, it would be nice to design an analysis dealing with the (multi-)sets of array cell contents.

A longer term perspective would be to generalize our abstract values for other uses: an array is a special case of function, so an interesting question is whether, using similar principles, we could design an abstract domain for expressing function properties.

Acknowledgments: We are indebted to Tom Reps and Bertrand Jeannet for helpful discussions, and to the anonymous referees for their constructive comments.

References

[1] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. Preliminary version appears in the Proc. of 5th LICS, 1990.

[2] S. Bardin, J. Leroux, and G. Point. Tool presentation: Fast extended release. In *18th Conf. Computer Aided Verification (CAV'2006)*, pages 63–66, Seattle (Washington), 2006. LNCS 4144, Springer-Verlag.

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In J. Ferrante and K. S. McKinley, editors, *PLDI 2007*, pages 300–309. ACM, 2007.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI 2003, ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation*, pages 196–207, San Diego (Ca.), June 2003.

[5] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *Computer Aided Verification (CAV 2006)*, pages 517–531. LNCS 4144, Springer Verlag, July 2006.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI 06*, pages 427–442. LNCS 3855, Springer Verlag, 2006.

[7] R. C. Claris and J. Cortadella. Verification of parametric timed circuits using octahedra. In *Designing correct circuits, DCC'04*, Barcelona, March 2004.

[8] P. Cousot. Verification by abstract interpretation. In N. Dershowitz, editor, *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268, Taormina, Italy, June 29 – July 4 2003. © Springer-Verlag, Berlin, Germany.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[11] D. L. Dill. Timing assumptions and verification of finite state concurrent systems. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL 2002*, pages 191–202. ACM, 2002.

[13] D. Gopan. *Numeric program analysis techniques with applications to array analysis and library summarization*. PhD thesis, Computer Science Department, University of Wisconsin, Madison, WI, August 2007.

[14] D. Gopan, F. Di Maio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS'04*, pages 512–529, Barcelona, 2004.

[15] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL'2005*, pages 338 – 350, Long Beach, CA, 2005.

[16] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In G. C. Necula and P. Wadler, editors, *POPL 2008*, pages 235–246. ACM, 2008.

[17] C. A. R. Hoare. Proof of a program: Find. *CACM*, 14(1):39–45, 1971.

[18] R. Iosif, P. Habermehl, and T. Vojnar. What else is decidable about arrays? In R. Amadio, editor, *FOSSACS 2008*. LNCS, Springer Verlag, 2008.

[19] R. Jhala and K. L. McMillan. Array abstractions from proofs. In W. Damm and H. Hermanns, editors, *CAV 2007*, pages 193–206. LNCS 4590, Springer Verlag, 2007.

[20] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In R. Alur and D. Peled, editors, *CAV 2004*, pages 135–147. LNCS 3114, Springer Verlag, 2004.

[21] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt Jr. and F. Somenzi, editors, *CAV 2003*, pages 141–153. LNCS 2725, Springer Verlag, 2003.

[22] F. Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 226–235, New York, NY, USA, 1992. ACM.

[23] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[24] M. Péron and N. Halbwachs. An abstract domain extending Difference-Bound Matrices with disequality constraints. In B. Cook and A. Podelski, editors, *8th International Conference on Verification, Model-checking, and Abstract Intepretation, VMCAI'07*, Nice, France, January 2007.