Analyzing array manipulating programs by program transformation

J. Robert M. Cornish[1], Graeme Gange[1], Jorge A. Navas[2], Peter Schachte[1], Harald Søndergaard[1], and Peter J. Stuckey[1]

[1] Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia
j.cornish@student.unimelb.edu.au
{gkgange,schachte,harald,pstuckey}@unimelb.edu.au
[2] NASA Ames Research Center, Moffett Field, CA 94035, USA
jorge.a.navaslaserna@nasa.gov

Abstract. We explore a transformational approach to the problem of verifying simple array-manipulating programs. Traditionally, verification of such programs requires intricate analysis machinery to reason with universally quantified statements about symbolic array segments, such as "every data item stored in the segment A[i] to A[j] is equal to the corresponding item stored in the segment B[i] to B[j]." We define a simple abstract machine which allows for set-valued variables and we show how to translate programs with array operations to array-free code for this machine. For the purpose of program analysis, the translated program remains faithful to the semantics of array manipulation. Based on our implementation in LLVM, we evaluate the approach with respect to its ability to extract useful invariants and the cost in terms of code size.

1 Introduction

We revisit the problem of automated discovery of invariant properties in simple array-manipulating programs. The problem is to extract interesting properties of the contents of one-dimensional dynamic arrays (by dynamic we mean arrays whose bounds are fixed at array variable creation time, but not necessarily at compile time). We follow the *array partitioning* approach proposed by Gopan, Reps, and Sagiv [9] and improved by Halbwachs and Péron [11]. This classical approach uses two phases. In a first phase, a program analysis identifies all (potential) symbolic *segments* by analyzing all array accesses in the program. Each segment corresponds to an interval I_k of the array's full index domain, but its bounds are symbolic, that is, bounds are *index expressions*. For example, the analysis may identify three relevant segments $I_1 = [0, \ldots, i-1]$, $I_2 = [i]$, and $I_3 = [i+1, \ldots, n-1]$. After this the original array A is considered partitioned into segments A_{I_k} corresponding to the identified segments and each segment is replaced with a *summary variable* a_k. In the second phase, the analysis aims at discovering properties $\psi(a_k)$ on each summary variable a_k such that

$$\forall\, \ell \in I_k(\psi(a_k) \Rightarrow \psi(A[\ell])) \tag{1}$$

By partitioning arrays into segments, the analysis can produce stronger separate analyses for each segment rather than a single weaker combined result for the whole array. In particular, we can identify *singleton* segments (A_{I_2} in the example) and translate array writes to these as so-called strong updates. A *strong update* benefits from the fact that the old content of the segment is eliminated by the update, so the new content replaces the old. For a segment that may contain multiple elements, an assignment to an array cell may leave some content unchanged, so a *weak update* must be used, that is, we must use a lattice-theoretic "join" of the new result and the old result associated with ℓ.

Although very accurate, array partitioning methods have their drawbacks. Partitioning can be prohibitively expensive, with a worst-case complexity of $O(n!)$, where n is the number of program variables. Moreover, partitioning must be done before the array content analysis phase that aims at inferring invariants for the form (1), which could be less precise than doing both simultaneously [5]. To mitigate this problem, the index analysis, used to infer the relevant symbolic intervals, is run twice: once during the segmentation phase and again during the array content analysis, which needs it to separate the first fixed point iteration from the rest. In the more sophisticated approach of Halbwachs and Péron [11, 16], the transfer functions are much more complex and a concept of "shift variables" is used, representing translation (in the geometric sense) of segments. This is not easily implemented using existing abstract interpretation libraries.

Contribution. We present a program transformation that allows scalar analysis techniques to be applied to array manipulating programs. As in previously proposed array analyses [9, 11, 16], we partition arrays into segments whose contents are treated as sets rather than sequences. To maintain the relationship among corresponding elements of different arrays, we abstract the state of all arrays within a segment to a set of vectors, one element per array. Thus we transform an array manipulating program into one that manipulates scalars and sets of vectors. A major challenge in this is to encode the disjunctive information carried by each array segment. We propose a technique that splits basic blocks. It has been implemented using the LLVM framework.

Importantly, a program transformation approach allows the separation of concerns: existing analyses based on any scalar abstract domains can be used directly to infer array content properties, even interprocedurally. While other approaches lift a scalar abstract domain to arrays by lifting each transfer function, our approach uses existing transfer functions unchanged, only requiring the addition of two simple transfer functions easily defined in terms of operations that already exist for most domains. The approach is also parametric in the granularity of array index sets, ranging from array smashing [2] to more precise (and expensive) instances. When we go beyond array smashing, the transformational approach inherits the exponential search cost present in the Halbwachs/Péron approach, as for some programs P, the transformed programs P' are exponentially larger than P. However, for simple array-based sort/search programs [9, 11], a transformational approach is perfectly affordable, in particular as we can capitalize on code optimization support offered by the LLVM infrastructure.

Instructions	I	$\rightarrow$	$v_1 = constant \mid v_1 = \circ\, v_2 \mid v_1 = v_2 \diamond v_3 \mid \mathsf{A}$
Array assignments	A	$\rightarrow$	$v_1 = arr[v_2] \mid arr[v_1] = v_2$
Jumps	J	$\rightarrow$	$\texttt{If}\ (v_1 \bowtie v_2)\ label_1\ label_2 \mid \texttt{Jmp}\ label \mid error \mid end$
Blocks	B	$\rightarrow$	$label : \mathsf{I}^*\ \mathsf{J}$
Programs	P	$\rightarrow$	B^+

Fig. 1. A small control-flow language with array expressions

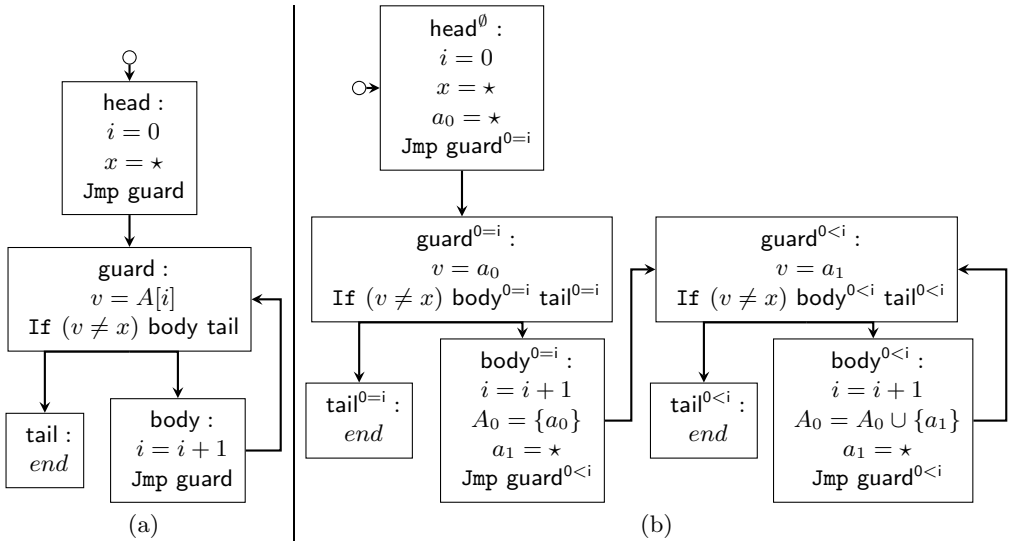


Fig. 2. (a) An array program fragment and (b) the corresponding set-machine program.

2 Source and target language

Our implementation uses LLVM IR as source and target language. However, as the intricacies of static single-assignment (SSA) form obscure, rather than clarify, the transformation, we base our presentation on a small traditional control flow language, whose syntax is given in Fig. 1. We shall usually shorten "basic block" to "block" and refer to a block's label as its identifier.

Source. Each block is a (possibly empty) sequence of instructions, followed by a (conditional) jump. Arithmetic unary and binary operators are denoted by $\circ$ and $\diamond$ respectively, and logical operators by $\bowtie$. We assume that there is a fixed set of arrays $\{A_1, \ldots, A_k\}$, which have global scope (and do not overlap in memory). The semantics is conventional and not discussed here. Fig. 2(a) shows an example program in diagrammatic form.

Target. The abstract machine we consider operates on variables over two kinds of domains: standard scalar types, and sets of vectors of length k, where k is the number of arrays in the source. The scalar variables represent scalars of the source program, including index variables, as well as singleton array segments; sets of vectors represent non-singleton segments of all extant arrays. Let $\mathbf{V}$ be the

$$
\begin{array}{lll}
\text{Instructions} & \mathsf{I} & \rightarrow \; v_1 = \mathit{constant} \mid v_1 = \circ\, v_2 \mid v_1 = v_2 \diamond v_3 \mid \mathsf{S} \\
\text{Set operations} & \mathsf{S} & \rightarrow \; S_1 = \mathsf{nondet\text{-}subset}(S_2) \mid S_1 = S_2 \cup S_3 \;\mid\; (v_1, \ldots, v_k) = \mathsf{nondet\text{-}elt}(S_1) \\
\text{Jumps} & \mathsf{J} & \rightarrow \; \texttt{If}\ (v_1 \bowtie v_2)\ \mathit{label}_1\ \mathit{label}_2 \mid \texttt{Jmp}\ \mathit{label} \mid \mathit{error} \mid \mathit{end} \\
\text{Blocks} & \mathsf{B} & \rightarrow \; \mathit{label} : \mathsf{I}^*\ \mathsf{J} \\
\text{Programs} & \mathsf{P} & \rightarrow \; \mathsf{B}^+
\end{array}
$$

Fig. 3. Control-flow language for the set machine.

$$
\begin{array}{rl}
\mathscr{S}[\![S_1 = S_2 \cup S_3]\!]\ \langle \sigma, \rho \rangle & = \langle \sigma, \rho[S_1 \mapsto \rho(S_2) \cup \rho(S_3)] \rangle \\
\mathscr{S}[\![S_1 = \mathsf{nondet\text{-}subset}(S_2)]\!]\ \langle \sigma, \rho \rangle & = \langle \sigma, \rho[S_1 \mapsto s] \rangle, s \subseteq \rho(S_2), s \neq \emptyset \\
\mathscr{S}[\![(v_1, \ldots, v_k) = \mathsf{nondet\text{-}elt}(S_1)]\!]\ \langle \sigma, \rho \rangle & = \langle \sigma[v_1 \mapsto x_1, \ldots, v_k \mapsto x_k], \rho \rangle, (x_1, \ldots, x_k) \in \rho(S_1)
\end{array}
$$

Fig. 4. Semantics for set manipulating operations.

set of scalar variables and $\mathbf{S}$ be the set of vector set variables. The runtime state of the machine is given by a pair $\langle \sigma, \rho \rangle$ consisting of a variable store $\sigma : \mathbf{V} \rightarrow \mathbb{Z}$, and a set store $\rho : \mathbf{S} \rightarrow \mathcal{P}(\mathbb{Z}^k)$.

A control flow language for set machine programs is given in Fig. 3. Arithmetic and logical operations affect only the variable store σ; the semantic rules for these operations are standard. The set machine also has set operations union ($\cup$), subset (nondet-subset) and element_of (nondet-elt). Fig. 4 gives their semantic rules, distinguishing scalar variables v and (vector) set variables S.

The union update $S_1 = S_2 \cup S_3$ maps S_1 to the union of values of S_2 and S_3. The subset and element operations are non-deterministic: executing $S_1 = \mathsf{nondet\text{-}subset}(S_2)$ assigns to S_1 *some* non-empty subset of elements from S_2, but makes no guarantee as to which elements are selected. Similarly, the element operation $(v_1, \ldots, v_k) = \mathsf{nondet\text{-}elt}(S_1)$ nondeterministically selects some element of vector set S_1 to load into $v_1, \ldots, v_k$.

Translation. Fig. 2(a)'s program scans an array for the first occurrence of value x, assumed to occur in A. The constraint $A[i] = x \wedge \forall\ k \in [0, i)\ (A[k] \neq x)$ is the desired invariant at tail. A corresponding array-free program is given in Fig. 2(b). The example illustrates some key features. Each contiguous array segment is represented by a set variable A_i. Each original block is duplicated for each feasible ordering of *interesting* variables. In the initial ordering $(0 = i)$ the only interesting segment is $A[0]$, represented as the singleton a_0; the read $v = A[i]$ is replaced by an assignment $v = a_0$. At $\mathsf{guard}^{0<i}$, $A[i]$ is represented by a_1 so the read is replaced by $v = a_1$. When i is updated at $\mathsf{body}^{0=i}$, the previous singleton a_0 becomes part of an "aggregate" segment $A[0, i-1]$. We then transform singleton a_0 to set A_0 and introduce a new singleton a_1 (representing $A[i]$ in the updated ordering). Similarly, when we update i in $\mathsf{body}^{0<i}$, segments $A[0, i-1]$ and $A[i]$ are merged (yielding $A_0 = A_0 \cup \{a_1\}$), and a new singleton a_1 is introduced. Consider the resulting concrete set-machine states. At $\mathsf{tail}^{0=i}$, we have $a_0 = x$, corresponding to $A[0] = x$ in the original program. At $\mathsf{tail}^{0<i}$, we find $x \notin A_0$ and $a_1 = x$. These correspond, respectively, to array invariants $\forall \ell \in [0, i-1]\ .\ A[\ell] \neq x$ and $A[i] = x$ in the original program.

3 From scalar to set machine transfer functions

We now show how to lift a scalar domain for use by set machines. Essentially, we use a scalar variable to approximate each component of each set; approximation of set-machine states can then be obtained by grouping states by values of the (original) scalar variables. Essentially, we approximate a set-machine state $\langle \sigma, \rho \rangle$ with set variables $\{S_1, \ldots, S_m\}$ by a set of scalar states $\{S_1^\diamond, \ldots, S_m^\diamond\}$ representing the possible results of selecting some element from each set:

$$\alpha^\diamond(\langle \sigma, \rho \rangle) = \{\sigma \cup \{S_1^\diamond \mapsto y_1, \ldots, S_m^\diamond \mapsto y_m\} \mid y_1 \in \rho(S_1), \ldots, y_m \in \rho(S_m)\}$$

Transfer functions for set operations then operate over the *universe* of possible states, rather than apply element-wise to each state.

Example 1. Consider a program with one scalar variable x, and one set variable S, with initial state $\langle \{x \mapsto 0\}, \{S \mapsto \{1, 2\}\} \rangle$. If we introduce a scalar variable y that selects a value from S via a we have two possible states:

$$\langle \{x \mapsto 0, y \mapsto 1\}, \{S \mapsto \{1, 2\}\} \rangle$$
$$\langle \{x \mapsto 0, y \mapsto 2\}, \{S \mapsto \{1, 2\}\} \rangle$$

If we represent S by scalar variable $S^\diamond$, we have initial states $\langle \{x \mapsto 0, S^\diamond \mapsto 1\} \rangle$ and $\langle \{x \mapsto 0, S^\diamond \mapsto 2\} \rangle$. When we wish to select a value for y, it is chosen nondeterministically from the possible values of $S^\diamond$, resulting in the states:

$$\langle \{x \mapsto 0, y \mapsto 1, S^\diamond \mapsto 1\} \rangle,\ \langle \{x \mapsto 0, y \mapsto 1, S^\diamond \mapsto 2\} \rangle$$
$$\langle \{x \mapsto 0, y \mapsto 2, S^\diamond \mapsto 1\} \rangle,\ \langle \{x \mapsto 0, y \mapsto 2, S^\diamond \mapsto 2\} \rangle$$

If we group states with equal values of x and y, we can see that these correspond to the final states of the original set-machine fragment. □

Note that this is an (over-)approximation. We can only infer the set of values that *may* be elements of S—this representation cannot distinguish sets of elements which may occur together, nor the cardinality of S. For example, assume we have possible set-machine states $\langle \emptyset, \{S \mapsto \{1\}\} \rangle$ and $\langle \emptyset, \{S \mapsto \{2\}\} \rangle$. The scalar approximation is $\langle \{S^\diamond \mapsto 1\},\ \{S^\diamond \mapsto 2\} \rangle$ which covers the feasible set-machine states, but also includes $\langle \emptyset, \{S \mapsto \{1, 2\}\} \rangle$, which is not feasible. More generally, if the set φ of concrete states allows sets $S \mapsto X_1, \ldots, S \mapsto X_k$, we have:

$$\forall X\ (X \subseteq X_1 \cup \ldots \cup X_k \Rightarrow (S \mapsto X) \in \gamma \circ \alpha(\varphi))$$

Consider a (not necessarily numeric) abstract domain $\mathcal{A}$, with meet ($\sqcap$), join ($\sqcup$) and rename operations, as well as a transfer function $\mathscr{F} : \mathsf{I} \to \mathcal{A} \to \mathcal{A}$ for the scalar fragment of the language. The rename operation constructs a new state where each variable x_i is replaced with y_i (then removes the existing bindings of x_i). Formally, the concrete semantics of `rename` is given by

$$\texttt{rename}(\sigma, [x_1, \ldots, x_k], [y_1, \ldots, y_k]) = \sigma \left[\begin{array}{c} y_1 \mapsto \sigma(x_1), \ldots, y_k \mapsto \sigma(x_k), \\ x_1 \mapsto \star, \ldots, x_k \mapsto \star \end{array} \right]$$

For each set variable S, we introduce k scalar variables $[s^1, \ldots, s^k]$ denoting the possible values of each vector in S. We then extend $\mathscr{F}$ to set operations as shown in Fig. 5.

$$\mathscr{F}[\![S_1 = \mathsf{nondet\text{-}subset}(S_2)]\!]\ \varphi = \varphi \sqcap \mathtt{rename}(\varphi, [s_2^1, \ldots, s_2^k], [s_1^1, \ldots, s_1^k])$$
$$\mathscr{F}[\![S_1 = S_2 \cup S_3]\!]\ \varphi = \begin{pmatrix} \mathscr{F}[\![S_1 = \mathsf{nondet\text{-}subset}(S_2)]\!]\ \varphi \\ \sqcup\ \mathscr{F}[\![S_1 = \mathsf{nondet\text{-}subset}(S_3)]\!]\ \varphi \end{pmatrix}$$
$$\mathscr{F}[\![(v_1, \ldots, v_k) = \mathsf{nondet\text{-}elt}(S_1)]\!]\ \varphi = \varphi \sqcap \mathtt{rename}(\varphi, [s_1^1, \ldots, s_1^k], [v_1^1, \ldots, v_1^k])$$

Fig. 5. Extending the transfer function for scalar analysis to set operations

4 Orderings

The transformation relies on maintaining a single ordering of index variables at each transformed block. We now discuss such total orderings.

Our goal is to partition the array index space $(-\infty, \infty)$ into contiguous regions bounded by index variables. For index variables i and j, we need to be able to distinguish between the cases where $i < j$, $i = j$ and $i > j$. However, this is not enough; if we assign $A[i] = x$, but only know that $i < j$, we cannot distinguish between the cases $i = j - 1$ (every element between i and j is x) and $i < j - 1$ (there are additional elements with some other property). So, for index variables i and j, we choose to distinguish these five cases:

$$\boxed{i+1<j} \qquad \boxed{i+1=j} \qquad \boxed{i=j} \qquad \boxed{i=j+1} \qquad \boxed{i>j+1}$$

For convenience in expressing these orderings, we will introduce for each index variable i a new term i^+ denoting the value $i+1$, and for a set of index variables $\mathbf{I}$ we will denote by $\mathbf{I}^+$ the augmented set $\mathbf{I} \cup \{v^+ \mid v \in \mathbf{I}\}$. We can then define a total ordering of a set of index variables $\mathbf{I}$ to be a sequence of sets $[B_1, \ldots, B_k]$, $B_s \subseteq \mathbf{I}^+$, such that the B_s's cover $\mathbf{I}^+$, are pairwise disjoint, and satisfy $i \in B_s \Leftrightarrow i^+ \in B_{s+1}$.

The meaning of the ordered list $\pi = [B_1, B_2, \ldots, B_k]$ is parameterised by the value of program variables involved, that is, it depends on a store σ. The meaning is: $[\![\pi]\!](\sigma) \equiv$

$$\bigwedge_{s,t \in [1..k]} (\forall e, e' \in B_s\ (\sigma(e) = \sigma(e')) \wedge \forall e \in B_s\ \forall e' \in B_t\ (s < t \rightarrow \sigma(e) < \sigma(e')))$$

An ordering π (plus virtual bounds $\{-\infty, \infty\}$) partitions the space of possible array indices into contiguous regions, given by $[\sigma(e), \sigma(e'))$ for $e \in B_i, e' \in B_{i+1}$. For any index variable i, a segment containing i^+ in the right bound is necessarily a singleton segment; all other segments are considered aggregate.

When a new index variable k enters scope, several possible orderings may result. Fig. 6(c) gives a procedure for enumerating them. When an index variable k leaves scope, computing the resulting ordering consists simply of eliminating k and k^+ from π, and discarding any now-empty sets. Assignment of an index variable is handled as a removal followed by an introduction.[3]

[3] If the assigned index variable appears in the expression, we assign the index to a temporary variable, and replace the index with the temporary in the expression.

We can discard any ordering that arranges constants in infeasible ways, such as $4 < 3$. If we have performed some scalar analysis on the original program, we need only generate orderings which are consistent with the analysis results.

5 The transformation

We now detail the transformation from an array manipulating program to a set-machine program, with respect to a fixed set of *interesting* segment bounds. Section 6 covers the selection of these bounds. Intuitively, the goal of the transformation is to partition the array into a collection of contiguous segments, such that each array operation uniquely corresponds to a singleton segment. Each singleton segment is represented by a tuple of scalars; each non-singleton segment is approximated by a set variable. There are two major obstacles to this. First, a program point does not typically admit a unique ordering of a given set of segment bounds; second, as variables are mutated in the program, the correspondence between concrete indices and symbolic bounds changes.

The transformation resolves this by replicating basic blocks to ensure that, at any program point, a unique partitioning of the array into segments is identifiable. Any time a segment-defining variable is modified, introduced or eliminated, we emit statements to distinguish the possible resulting partitions, and duplicate the remainder of the basic block for each case. For each partition, we also emit set operations to restore the correspondence between set variables and array segments, using `nondet-elt` and `nondet-subset` when a segment is subdivided, and $\cup$, when a boundary is removed, causing segments to be merged. This way every array read/write in the resulting program can be uniquely identified with a singleton segment. As singleton sets are represented by tuples of scalars, we can finally eliminate array operations, replacing them with scalar assignments.

In the following, we assume the existence of functions next_block, which allocates a fresh block identifier, and push_block, which takes an identifier, a sequence of statements and a branch, and adds the resulting block to the program. We also assume that there is a mutable global table T mapping block identifier and index variable ordering pairs $\langle id, \pi \rangle$ to ids, used to store previously computed partial transformations, and an immutable set $\mathcal{I}$ of segment bound variables and constants. The function get_block takes a block identifier, and returns the body of the corresponding block. The function vars returns the set of variables appearing lexically in the given expression. The function find_avar gives the variable name to which a given array and index will be translated, given an ordering.

Fig. 6 gives the transformation. Procedure transform takes a block and transforms it, assuming a given total ordering π of the index variables. It is called once with the initial block of each function and an ordering containing only the constants in the index set. As there are finitely many $\langle id, \pi \rangle$ combinations, and each pair is constructed at most once, this process terminates.

The core of the transformation is done by a call to transform_body(B, π, id, ss). Here B is the portion of the current block to be transformed and π the current ordering. id and ss hold the identifier and body of the partially-transformed

```
% Check if the block has already been transformed
% under π. If not, transform it.
transform(id, π)
    if ((id, π) ∈ T)
        return T[(id, π)]
    id_t := next_block()
    T := T[(id, π) ↦ id_t]
    (stmts, br) := get_block(id)
    transform_body((stmts, br), π, id, [])
    return id_t
% Evaluate a branch.
transform_body(([], Jmp b), π, id, ss)
    id_b := transform(b, π)
    push_block(id, ss, Jmp id_b)

transform_body(([], If l then t else f), π, id, ss)
    if vars(l) ⊆ I
        dest := if eval(l, π) then t else f
        id_dest := transform(dest, π)
        push_block(id, ss, Jmp id_dest)
    else
        id_t := transform(t, π)
        id_f := transform(f, π)
        push_block(id, ss, If l then id_t else id_f)
% (Potentially) update an index.
transform_body(([x = expr|stmts], br), π, id, ss)
    if x ∈ I
        split_transform(x, (stmts, br), π, id, ss ::[x = expr])
    else
        transform_body((stmts, br), π, id, ss ::[x = expr])
% Transform an array read...
transform_body(([x = A[i]|stmts], br), π, id, ss)
    A_i := find_avar(π, A, i)
    transform_body((stmts, br), π, id, ss ::[x = A_i])
% or an array write.
transform_body(([A[i] = x|stmts], br), π, id, ss)
    A_i := find_avar(π, A, i)
    transform_body((stmts, br), π, id, ss ::[A_i = x])
```

(a) The top-level transformation process

```
split_transform(x, (stmts, br), π, id, ss)
    Π' := feasible_orders(π, x)
    split_rec(x, Π', (stmts, br), π, id, ss)

split_rec(x, [π'], (stmts, br), π, id, ss)
    asts := remap_avars(π, π')
    transform_body((stmts, br), π', id, ss :: asts)

split_rec(x, [π'|Π'], (stmts, br), π, id, ss)
    id_π' := next_block()
    id_Π' := next_block()
    cond := ord_cond(x, π')
    push_block(id, ss, If cond then id_π' else id_Π')
    asts := remap_avars(π, π')
    transform_body((stmts, br), π', id_π', asts)
    split_rec(x, Π', (stmts, br), π, id_Π', [])
```

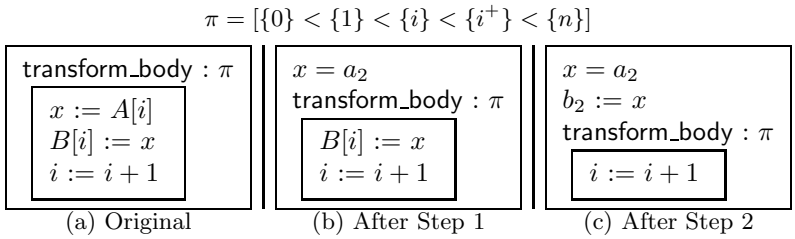
(b) Fan-out of a block when an index variable is changed

```
feasible_orders(k, π) : insert(k, π, [])

insert(k, [], pre) : return {pre ::{k} ::{k+}}
insert(k, [S_i|S], pre)
    low := insert+(k, [S_i | S], pre ::{k})
    high := insert(k, S, pre :: S_i)
    if ∃ x . x+ ∈ S_i
        return low ∪ high
    else
        return low ∪ high ∪
                insert+(k, S, pre ::(S_i ∪ {k}))

insert+(k, [], pre) : return {pre ::{k+}}
insert+(k, [S_i | S], pre)
    if ∃ x . x+ ∈ S_i
        return {pre ::(S_i ∪ {k+}) :: S}
    else
        return {pre ::(S_i ∪ {k+}) :: S} ∪
                {pre ::{k+} :: S_i :: S}
```

(c) Enumerating the possible total orderings upon introducing a new index variable k

Fig. 6. Pseudo-code for stages of the transformation process.

block. As a block is processed, instructions not involving index or array variables are copied verbatim into the transformed block. During the process, we ensure that each (transformed) statement is reachable under exactly one index ordering π. Singleton segments under π are represented by scalar variables, and aggregate segments by set variables. Array reads and writes are replaced with



(a) Original (b) After Step 1 (c) After Step 2

Fig. 7. Transformation of array reads and writes under ordering π. As the segment $[i, i^+]$ is a singleton, the array elements are represented as scalars.

accesses and assignments to the corresponding scalar or set variable, as determined by find_avar. Conditional branches whose conditions are determined by the current ordering are replaced by direct branches to the **then** or **else** part, as appropriate. Once no instructions remain to be transformed, the block id is emitted with body ss, together with the appropriate branch instruction.

Whenever an index variable is modified, the rest of the current block must be split, and the set variables must be updated accordingly. The rest of the block is then transformed under *each* possible new ordering π'. This is the job of split_transform shown in Fig. 6(b), while the job of feasible_orders in Fig. 6(c) is to determine the set of possible orders. The function ord_cond(x, π') generates logical expressions to determine whether the ordering π' holds, given that π previously held. ord_cond checks the position of both x and x^+. If x is part of a larger equivalence class in π, ord_cond generates the corresponding equality; otherwise, it checks that x is greater than its left neighbour; similarly, it checks that x^+ is in its class or less than its right neighbour. Fig. 6(b) shows the process of splitting a block upon introducing an index variable x.

5.1 Reading and writing

Transformation of array reads and writes is simple, if the array index is in the set **I** of index variables. Fig. 7(a–c) shows the step-by-step transformation of a block, under the specified ordering. After Step 1, reference $A[i]$ has been transformed to scalar a_2, since $\{i\}$ is a singleton. Similarly, Step 2 transforms $B[i]$ to b_2.

If the index of the read/write operation has been omitted, we must instead emit code to ensure the operation is dispatched to the correct set variable. The dispatch procedure is similar in nature to split_transform, as given in Fig. 6(c); essentially, we emit a series of branches to determine which (if any) of the current segments contains the read/write index. Once this has been determined, we apply the array operation to the appropriate segment. If the selected segment is a singleton, this is done exactly as in transform_body. For writes to an aggregate segment, we must first read some vector from the segment, substitute the element to be written, then merge the updated vector back into the segment.[4]

[4] Detailed pseudo-code for this is in Appendix A.

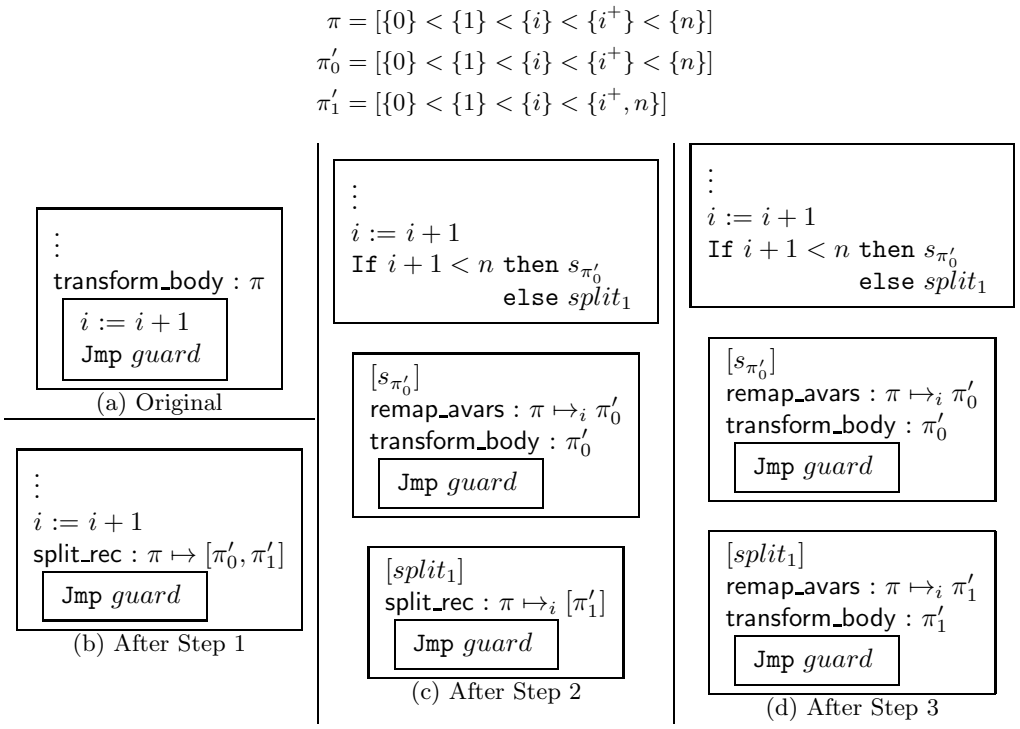


Fig. 8. Example of updating an index assignment. We assume an existing scalar analysis which has determined that, after $i = i + 1$, we have $1 < i < n$.

5.2 Index manipulation

The updating of index variables is the most involved part of the transformation, as we must emit code not only to determine the updated ordering π', but also to ensure the array segment variables are matched to the corresponding bounds. Fig. 8 illustrates this process, implemented by the procedure remap_avars, as it splits a block into three: one to test an index expression to determine what ordering applies, and one for each ordering. In the original code, ordering π applies, but following the assignment, either ordering π'_0 or π'_1 may apply. The test inserted by Step 2 distinguishes these cases, leaving only one ordering applicable to each of the $s_{\pi'_0}$ and $split_1$ blocks.

If we normalize index assignments such that for $k := E$, $k \notin E$, we can separate the updating of segment variables into two stages; first, computing intermediate segment variables A'_i after eliminating k from π, and then computing the new segment variables after introducing the updated value of k. Pseudo-code for these steps are given in Fig. 9(a) and 10(a). In practice, we can often eliminate many of these intermediate assignments, as segments not adjacent to the initial or updated values of k remain unchanged.

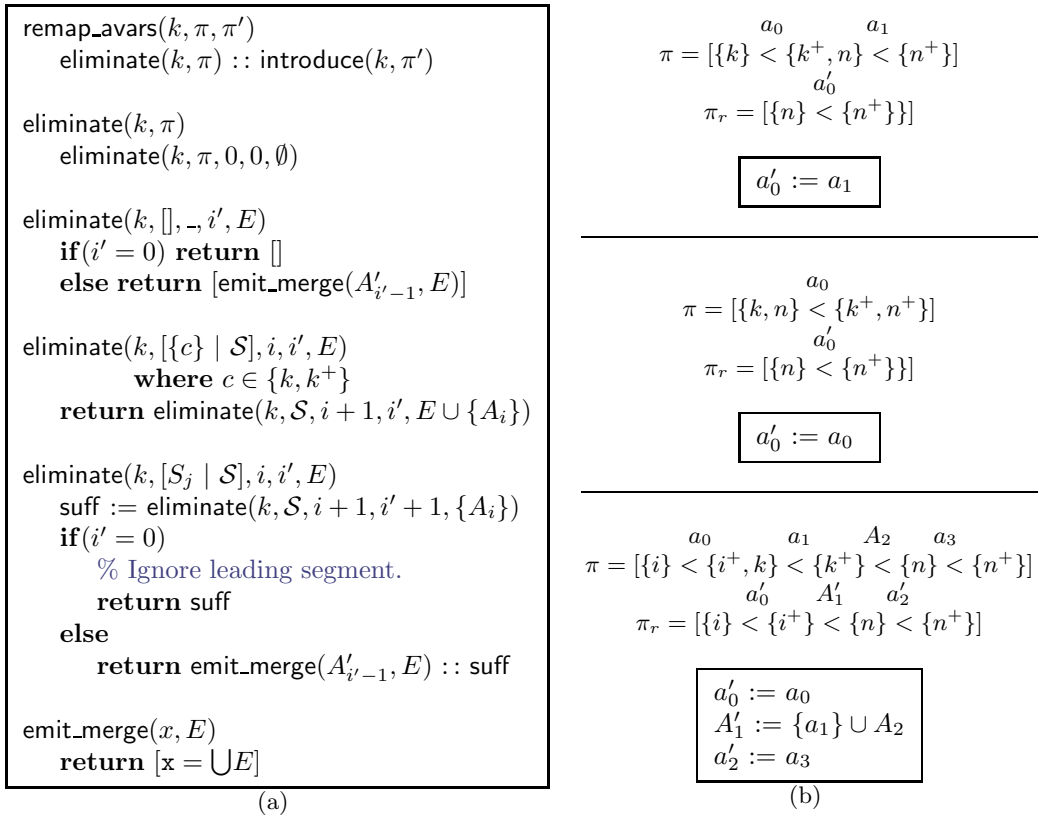


Fig. 9. (a) Algorithm for generating instructions to keep segment variables updated; (b) resulting assignments when k is eliminated from various orderings, also showing the remaining order π_r and scalar or set variables corresponding to each segment.

When we eliminate an index variable k from π, we merge segments that were bounded only by k or k^+. If k or k^+ appears alone at the very beginning or end of π, the segments are discarded entirely. If either appears alone between other variables in π, the segments on either side are merged to form a single segment. However, if k and k^+ are both equal to some other variables, the original segments are simply copied to the corresponding temporary variables. This is illustrated in Fig. 9(b).

The pseudo-code in Fig. 9 and 10 ignores the distinction between singleton and aggregate segments; the transformed operations differ slightly in the two cases. If we introduce a singleton segment into an aggregate segment, we select a single vector from the set ($(a', b', c') =$ nondet-elt(A)); if an aggregate segment is introduced, we emit a subset operation ($A' =$ nondet-subset(A)).

The procedure for injecting k into π behaves similarly. If k is introduced at either end of π, we introduce new segments with indeterminate values. If k

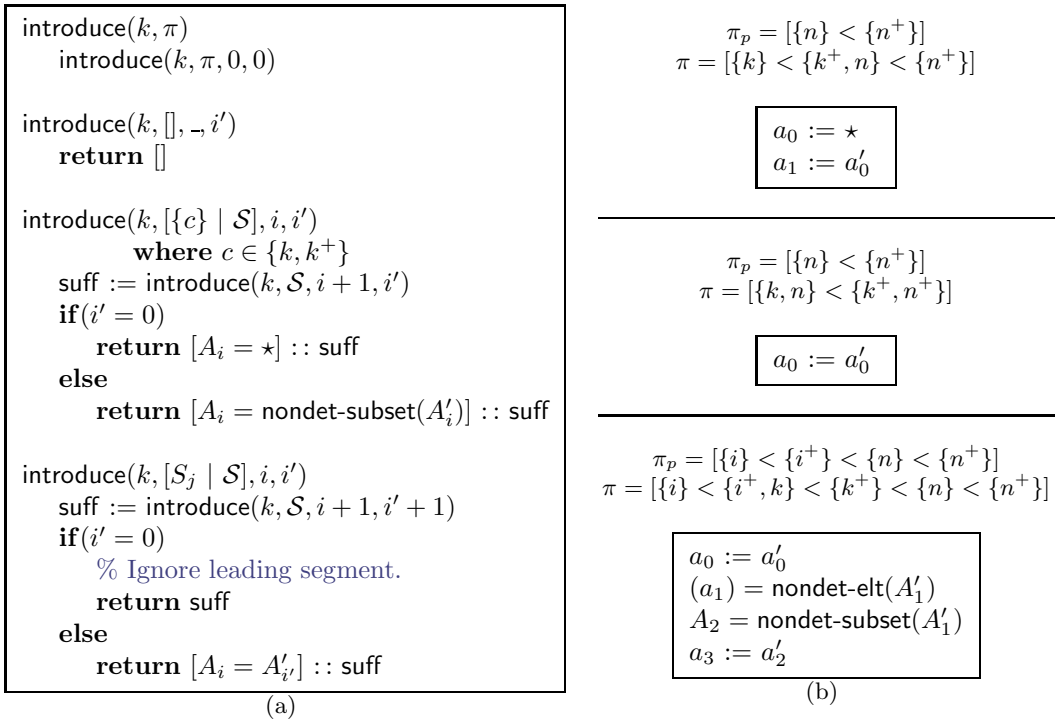


Fig. 10. (a) Generating instructions for (re-)introducing a variable k into a given ordering, and (b) the resulting assignments when k is introduced into various orderings. Note the difference between introducing singleton and aggregate segments.

is introduced somewhere within an existing segment, we introduce new child segments—each of which is a subset of the original segment.

5.3 Control flow

When transforming control flow, there are three cases we must consider:

1. Unconditional jumps
2. Conditional jumps involving some non-index variables
3. Conditional jumps involving *only* index variables

In cases (1) and (2), the transformation process operates as normal; we recursively transform the jump targets, and construct the corresponding jump with the transformed identifiers. However, when we have a conditional jump `If` $i \bowtie j$ `then` t `else` f where i and j are both index terms, the relationship between i and j is statically determined by the current ordering π. As a result, we can simply evaluate the condition $i \bowtie j$ under the ordering π, and use an unconditional branch to the corresponding block. This is illustrated in Fig. 11.

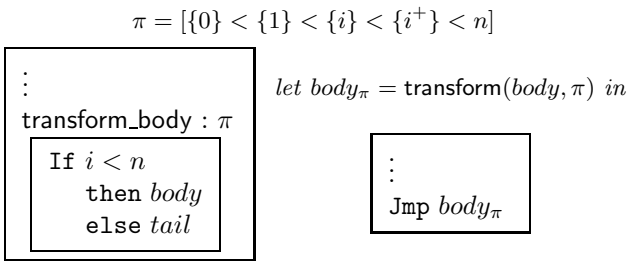


Fig. 11. Transforming a jump, conditional on index variables only, under ordering π

6 Selecting segment bounds

Until now we have assumed a pre-selected set of *interesting* segment bounds. The selection of segment boundaries involves a trade-off: we can improve the precision of the analysis by introducing additional segment bounds, but the transformed program grows exponentially as the number of segment bounds increases. As do [11], we can run a data-flow analysis to find the set of variables that may (possibly indirectly) be used as, or to compute, array indices. Formally, we collect the set $\mathbb{I}$ of variables and constants i occurring in these contexts:

$$A[i] \quad \text{where } A \text{ is an array} \tag{2}$$

$$i' = \mathsf{op}(i) \quad \text{where } i' \in \mathbb{I} \tag{3}$$

$$i = \mathsf{op}(i') \quad \text{where } i' \in \mathbb{I} \text{ and } i' \text{ is not a constant} \tag{4}$$

Any variable which does not satisfy these conditions can safely be discarded as a possible segment bound. For the experiments in Section 7 we used all elements of $\mathbb{I}$ as segment bounds (so $\mathbf{I} = \mathbb{I}$), which yields an analysis corresponding roughly to the approaches of [9, 11]. We could, however, discard some subset of $\mathbb{I}$ to yield a smaller, but less precise, approximation of the original program. The cases (3) and (4) are needed because of possible aliasing; this is particularly critical in an SSA-based language, as SSA essentially replaces mutation with aliasing. It is worth noting that these dependencies extend to aliases introduced prior to the relevant array operation, as in the snippet "$i := x; \ldots A[i] := k; \ldots y := x + 1;$"

7 Experimental evaluation

We have implemented our method using the LLVM framework, in two distinct transformation phases. In a first pass, transformation is done as described above, but without great regard for the size of the transformed program. At the same time, we also use a (polyhedral) scalar analysis of the original program (treating arrays as unknown value sources) to detect any block whose total ordering is infeasible. In the second pass, we prune these unreachable blocks away. As can be gleaned from Table 1, these measures reduce the complexity of the transformed program significantly.

To extract array properties from the corresponding invariants discovered in a transformed program, we require users to specify, at transformation time, the range of array segments that are of interest. In our implementation, this is described by a strict index inequality that must apply to segments in the range. For example, specifying $0 < n$ indicates that we are interested in invariants of the form $\forall \ell \ (0 \leq \ell < n \Rightarrow \psi(A_1[\ell], \ldots, A_k[\ell]))$, where $A_1, \ldots, A_k$ are the arrays in scope and ψ is some property. At the end of the transformation we use a newly created block to join all copies of the original exit block whose total ordering is consistent with the given range. The various scalar representations for each array segment, as well as other variables in scope in each copy, are merged together in phi nodes inside this final block. Properties discovered about the segment phi nodes then translate directly to properties about the corresponding array segments in the original program.

We have tested our method by running first the `polka` polyhedra domain [12] on the output of our transformation when applied to the programs given in Fig. 12. The interesting invariants that we infer are as follows (each property holds at the end of the corresponding function):

```
array_copy (int* A, int* B, int n) {
    int i;
    for (i = 0; i < n; i++)
        A[i] = B[i];
}
```

```
array_init (int* A, int n) {
    int i;
    for (i = 0; i < n; i++)
        A[i] = 5;
}
```

```
array_max (int* A, int n) {
    int i, max = A[0];
    for (i = 1; i < n; i++) {
        if (max < A[i])
            max = A[i] }
}
```

```
search (int* A, int key) {
    int i = 0;
    while (A[i] ≠ key)
        i++;
}
```

```
first_not_null (int* A, int n) {
    int i, s = n;
    for (i = 0; i < n; i++)
        if (s == n && A[i] ≠ 0)
            s = i;
}
```

```
sentinel (int* A, int n, int sent) {
    int i;
    A[n − 1] = sent;
    for (i = 0; A[i] ≠ sent; i++);
}
```

Fig. 12. Simple test programs

$$
\begin{array}{ll}
\text{array\_copy} & : \forall \ell \ (0 \leq \ell < n \Rightarrow A[\ell] = B[\ell]) \\
\text{array\_init} & : \forall \ell \ (0 \leq \ell < n \Rightarrow A[\ell] = 5) \\
\text{array\_max} & : \forall \ell \ (0 \leq \ell < n \Rightarrow A[\ell] \leq max) \\
\text{search} & : \forall \ell \ (0 \leq \ell < i \Rightarrow A[\ell] \neq key) \\
\text{first\_not\_null} & : \forall \ell \ (0 \leq \ell < s \Rightarrow A[\ell] = 0) \\
\text{sentinel} & : \forall \ell \ (0 \leq \ell < i \Rightarrow A[\ell] \neq sent)
\end{array}
$$

Fig. 12 shows test programs from related papers [9, 11]. Table 1 lists sizes of the original, transformed, and post-processed transformed versions (columns Original, Transformed, and Post-processed respectively), as well as the time to perform the transformation (column transf). Column `polka` shows the analysis time in seconds for running the polka polyhedra domain. `uva` is explained below.

Program	Original		Transformed		Post-processed		Running time (s)		
	blocks	insts.	blocks	insts.	blocks	insts.	transf.	`polka`	`uva`
array_copy	5	12	274	898	33	149	0.80	67.07	0.18
array_init	5	11	274	644	33	115	0.94	19.08	0.22
array_max	7	19	220	562	51	139	0.95	110.87	0.45
search	5	10	90	167	27	69	0.49	2.05	2.75
first_not_null	8	17	1057	2217	216	694	3.73	2378.35	4.85
sentinel	5	13	1001	1936	294	765	3.07	1773.01	4.97

Table 1. Sizes of transformed test programs and analysis time for `polka` and `uva`

Enhancing an existing analyzer. As a separate experiment we use IKOS [3], an abstract interpretation-based static analyzer developed at NASA. IKOS has been used successfully to prove absence of buffer overflows in Unmanned Aircraft Systems flight control software written in C. The latest (unreleased) version of IKOS provides an uninitialized variable analysis that aims at proving that no variable can be used without being previously defined, otherwise the execution of the program might result in undefined behaviour.

Currently IKOS is not sufficiently precise for array analysis. (Fig. 13), IKOS cannot deduce that $A[5]$ is definitely uninitialized at line 4. However, using the transformational approach, IKOS proves that $A[5]$ is definitely uninitialized. The problem is far from trivial; as Regehr [17] notes, `gcc` and `clang` (with *-Wuninitialized*) do not even raise warnings for this example, but stay completely silent.

```
    int array_init_unsafe (void) {
1:      int A[6], i;
2:      for (i = 0; i < 5; i++)
3:          A[i] = 1;
4:      return A[5];
    }
```

Fig. 13. Regehr's example [17]

We ran IKOS on the transformed version of array_init_unsafe. IKOS successfully reported a definite error at line 4 in 0.22 seconds. Conversely, transformation enabled IKOS to show that no array element was left undefined in the case of array_init. Finally we ran IKOS on the rest of the programs in Fig. 12. For the purpose of the uninitialized variable analysis we added loops to force each array to be treated as initialized, when appropriate. For the transformed version of array_copy, IKOS proved that A is definitely initialized after the execution of the loop. For the rest of the programs IKOS proved that the initialized array A is still initialized after the loops. Column `uva` in Table 1 shows the analysis time in seconds of the uninitialized variable analysis implemented in IKOS.

Note that the `polka` analysis does not eliminate out-of-scope variables. Our program transformation introduces many variables, and since `polka` incurs a super-linear per-variable cost, the overall time penalty is considerable. We expect to be able to greatly reduce the cost by utilising a projection operation and improving the fixed-point finding algorithm.

8 Related work

Amongst work on automated reasoning about array-manipulating code, we can distinguish work on *analysis* from work that focuses on *verification*. Our paper is concerned with the analysis problem, that is, how to use static analysis for automated generation of (inductive) code invariants. As mentioned in Section 1, we follow the tradition of applying abstract interpretation [4] to the array content analysis problem [5, 9, 11, 16]. Alternative array analysis methods include Gulwani, McCloskey and Tiwari's lifting technique [10] (requiring the user to specify *templates* that describe when quantifiers should be introduced), Kovács and Voronkov's theorem-prover based method [13], Dillig, Dillig and Aiken's fluid updates [7] (supporting points-to and value analysis but excluding relational analyses), and incomplete approaches based on dynamic analysis [8, 15].

Unlike previous work, we apply abstract interpretation to a transformed program in which array reads and writes have been translated away; any standard analysis, relational or not, can be applied to the resulting program, with negligible additional implementation cost.

There is a sizeable body of work that considers the *verification* problem for array-processing programs. Here the aim is to establish that given assertions hold at given program points. While abstract interpretation may serve this purpose (given a well-chosen abstract domain), more direct approaches are goal-directed, using assertions actively, to drive reasoning, rather than passively, as checkpoints. Many alternative techniques have been suggested for the verification of (sometimes restricted) array programs, including lazy abstraction [1], template-based methods [14], and, more closely related to the present paper, techniques that employ translation, for example to Horn clauses [6].

9 Conclusion

We have described a new abstract machine that supports set-valued variables and shown how array manipulating programs can be translated to array-free code for this machine. By compiling array programs for this machine, we are able to discover non-trivial universally quantified loop invariants, simply by analysing the transformed program using off-the-shelf scalar analysers. As an example of how this allows an existing analysis to be lifted to array programs in a straightforward manner, we have extended an uninitialised-variable analysis; Figure 13 showed the usefulness of this approach. The indisputable price for the ease of implementation is a potentially excessive size of the transformed program. However, much array-processing code tends to make simple array traversals and access, and the transformational approach is viable for more than just small programs. Future work includes performing the transformation lazily, to avoid generating unneeded blocks. This should significantly speed up the analysis.

Acknowledgements

This work was supported through ARC grant DP140102194.

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In N. Bjørner and A. Voronkov, editors, *LPAR'12*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.
3. G. Brat, J. A. Navas, N. Shi, and A. Venet. IKOS: A framework for static analysis based on abstract interpretation. In *SEFM'14*, 2014. To appear.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM Press, 1977.
5. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL'11*, pages 105–118. ACM Press, 2011.
6. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying array programs by transforming verification conditions. In K. L. McMillan and X. Rival, editors, *VMCAI'14*, volume 8318 of *LNCS*, pages 182–202. Springer, 2014.
7. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs weak updates. In A. Gordon, editor, *ESOP'10*, volume 6012 of *LNCS*, pages 246–266. Springer, 2010.
8. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.
9. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL'05*, pages 338–350. ACM Press, 2005.
10. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL'08*, pages 235–246. ACM Press, 2008.
11. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI'08*, pages 339–348. ACM Press, 2008.
12. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *CAV'09*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.
13. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In M. Chechik and M. Wirsing, editors, *FASE'09*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.
14. D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI'13*, volume 7737 of *LNCS*, pages 169–188, 2013.
15. T. V. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*, pages 760–770. IEEE, 2012.
16. M. Péron. *Contributions à l'analyse statique de programmes manipulant des tableaux*. PhD thesis, Université de Grenoble, 2010.
17. J. Regehr. Uninitialized variables. Web blog, `http://blog.regehr.org/archives/519`, accessed 18 June 2014.

Appendix A: Array operations with non-segment variables

Fig. 6(a) assumes that the index variable of every read or write is included in the set of segment bounds. Fig. 14 gives a revised version of transform_body which handles writes to indices that are not included in the set of segment bounds. When we transform a write to an index in the set of segment bounds (determined by the predicate is_idx), the transformation is as usual. Otherwise, we emit code to walk through the current set of segments, and apply the write operation to the appropriate one. The dispatch process is similar to the operation of split_transform, except that all leaves jump back to the continuation of the basic block after the write, rather than continuing under the modified ordering.

```
transform_body(([A[i] = x|stmts], br), π, id, ss)
    if is_idx(i)
        A_i := find_avar(π, A, i)
        transform_body((stmts, br), π, id, ss ::[A_i = x])
    else
        id' := next_block()
        transform_body((stmts, br), π, id', [])
        dispatch_write(A[i] = x, ϵ, π, id, ss, id')

dispatch_write(A[i] = x, sv, [], id, ss, id')
    push_block(id, ss, Jmp id')

dispatch_write(A[i] = x, s_<, [p, ...|π], id, ss, id')
    id_≥ := next_block()
    id_= := next_block()
    s_= := next_svar(s_<)
    s_> := next_svar(s_=)
    if s_< = ϵ
        push_block(id, If i < p then id' else id_≥, ss)
    else
        id_< := next_block()
        push_block(id, If i < p then id_< else id_≥, ss)
        push_block(id_<, Jmp id',
            [(v_1, ..., v_A, ..., v_k) ∈ s_<,
             s_< = s_< ∪ {(v_1, ..., x, ..., v_k)}])
    push_block(id_≥, If i = p then id_= else id_>, id_=, id_>)
    (..., v_A, ...) := s_=
    push_block(id_=, Jmp id', [v_A = x])
    dispatch_write(A[i] = x, s_>, π, id_>, [], id')
```

Fig. 14. Revised pseudo-code for transforming array writes, allowing for omitted indices. Array reads are transformed similarly.