Array Abstraction with Symbolic Pivots

Reiner Hähnle(✉), Nathan Wasser, and Richard Bubel

Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
{haehnle,wasser,bubel}@cs.tu-darmstadt.de

Abstract. We present a novel approach to automatically generate invariants for loops manipulating arrays. The intention is to achieve formal verification of programs over arrays without the need for user-specified loop invariants. Many loops iterate and manipulate collections. Finding useful, i.e., sufficiently precise invariants for those loops is a challenging task, in particular, if the iteration order is complex. Our approach partitions an array and provides an abstraction for each of these partitions. Symbolic pivot elements are used to compute the partitions. In addition we integrate a faithful and precise program logic for sequential (Java) programs with abstract interpretation using an extensible multi-layered framework to compute array invariants. The presented approach has been implemented. Results of experiments are reported.

Keywords: Loop invariant generation · Program verification · Abstract interpretation · Array abstraction

1 Introduction

This paper is dedicated to Frank S. de Boer on the occasion of his 60th birthday. I met Frank in ca. 2006 during a phase in my research where I had become unhappy with the progress in formal verification of concurrent software. I was looking for partners who would be willing to co-develop a formal concurrent modeling language amenable to scalable formal specification and verification. At that time Frank and Einar B. Johnsen had been working on the distributed modeling framework Creol and we decided to team up. From this contact eventually the EU project HATS evolved where we developed the cloud-aware, concurrent OO modeling language ABS as well as the ongoing EU project Envisage where we look at SLAs. Looking back on ten years of intense collaboration, joint publications, and countless bottles of good wine enjoyed together, I can say: it was a great ride and I am looking forward to continue it! I learned a lot from Frank: he is a true generalist with a vast and deep knowledge in formal methods, his energy and creativity are contagious (and, occasionally, exasperating) and, most importantly, there is always fun to be had! Thank you, Frank, for these years together and let's have a toast to many more to come! — RH.

The work has been funded by the DFG priority program 1496 *Reliably Secure Software Systems*.

E. Ábrahám et al. (Eds.): de Boer Festschrift, LNCS 9660, pp. 104–121, 2016.
DOI: 10.1007/978-3-319-30734-3_9

Deductive program analysis and verification must determine a trade-off between the complexity of the properties they ascertain, the precision of the analysis, i.e., the percentage of issued false warnings, and the degree of automation.

Improving automation for medium to complex properties by maintaining an acceptable degree of precision requires addressing one of the sources for interaction (or otherwise loss of precision). One kind of interaction derives from the elimination of quantifiers, another one is the provision of program annotations such as method contracts, loop invariants or assertions that serve as hints for the underlying theorem prover. Providing useful annotations, in particular, loop invariants is a time-consuming and difficult task, which requires experience in writing formal specifications on the part of the user. This hinders wide-spread adoption of formal verification in industry.

Here we focus on the automatic generation of loop invariants. We improve upon previous work [1] of some of the co-authors in which a theoretical framework was developed that integrates deductive reasoning and abstract interpretation. We extend this by a novel approach for automatic generation of invariants for loops that manipulate arrays. This loop invariant generation works by partitioning arrays automatically using a new concept to which we refer as *symbolic pivots*. A symbolic pivot expresses the symbolic value of a term (in particular an array index) at the end of every loop iteration. When these symbolic pivots have certain properties we can generate highly precise partitions. The content of array partitions is represented as an abstract value which describes the value of the partition's elements. An important feature is that the degree of abstraction, that is, the precision is adaptive.

Further, we integrate a faithful and precise program logic for sequential (Java) programs with abstract interpretation using an extensible multi-layered framework to compute array invariants. The presented approach has also been implemented as a proof of concept based on the KeY verification system [2].

2 Background

2.1 Program Logic

We introduce our program logic and calculus, and explain our integration of value-based abstraction based on previous work [1] by some of the authors.

We stress that our implementation works for nearly full sequential Java [2], but for readability we restrict ourselves here to a fragment with integer arrays as the only kind of objects. The program logic presented below extends the logic in [1] by an explicit heap model and array types.

Syntax. We work with a first order dynamic logic which is closely related to Java Card DL [2]. Its signature is a collection of the symbols that can be used to construct formulas:

Definition 1 (Signature). *A* signature Σ *is a tuple* $((\mathcal{T}, \preceq), \mathcal{P}, \mathcal{F}, \mathcal{PV}, \mathcal{V})$ *consisting of a set of types* $\mathcal{T}$ *together with a type hierarchy* $\preceq$*, predicates* $\mathcal{P}$*, functions* $\mathcal{F}$*, program variables* $\mathcal{PV}$ *and logical variables* $\mathcal{V}$*. Types contain at least* $\top$*,* `Heap`, `LocSet`, `int` *and* `int[]` *with* $\top$ *being the top element and the other types ordered directly below* $\top$*.*

Our logic consists of terms *Trm* (write Trm_T for terms of type T), formulas *For*, programs *Prog* and updates *Upd*. Besides some extensions we elaborate on below, terms and formulas are defined as in standard first-order logic. Importantly, there is a difference between logical variables and program variables: both are terms, but logical variables must not occur in programs and can be bound by a quantifier. On the other hand, program variables can occur in programs, but cannot be bound by a quantifier. Syntactically, program variables are flexible function constants, whose value can be changed by executing a program.

Updates are discussed in [2] and can be viewed as generalized explicit substitutions. The grammar of updates is: $\mathcal{U} ::= (\mathcal{U} \,\|\, \mathcal{U}) \;\mid\; \mathtt{x} := t$ where $\mathtt{x} \in \mathcal{PV}$ and t is a term of the same type or subtype as $\mathtt{x}$. Updates can be applied to terms and formulas: given a term t then $\{\mathcal{U}\}t$ is also a term (analogous for formulas). The only other non-standard operator for terms and formulas in our logic is the conditional term: let φ be a formula and ξ_1, ξ_2 are both terms of compatible type or are both formulas, then *if* (φ) *then* (ξ_1) *else* (ξ_2) is also a term or formula.

There is a modality called box $[\cdot]\cdot$ which takes a program as first parameter and a formula as second parameter. Intuitively the meaning of $[\mathtt{p}]\phi$ is that *if* program `p` terminates without throwing an exception then in its final state the formula ϕ holds (our programs are deterministic). Thus the box modality expresses partial correctness. The formula $\phi \rightarrow [\mathtt{p}]\psi$ has the exact same meaning as the Hoare triple $\{\phi\}$ `p` $\{\psi\}$. In contrast to Hoare logic, dynamic logic allows nested modalities. The grammar for programs is:

$$\mathtt{p} ::= \mathtt{x} = t \mid \mathtt{x}[t] = t \mid \mathtt{p};\mathtt{p} \mid \mathtt{skip} \mid \mathtt{if}\,(\phi)\,\{\mathtt{p}\}\;\mathtt{else}\;\{\mathtt{p}\} \mid \mathtt{while}\,(\phi)\,\{\mathtt{p}\}$$

where $\mathtt{x} \in \mathcal{PV}$, t, φ are terms/formulas. Syntactically valid programs are well-typed and do not contain logic variables, quantifiers or modalities.

The program `skip` should have no effect. We write `if` (φ) $\{\mathtt{p}\}$ as an abbreviation for `if` (φ) $\{\mathtt{p}\}$ `else` $\{$ `skip` $\}$.

Semantics. Terms, formulas and programs are evaluated with respect to a first order structure.

Definition 2 (First Order Structure, Variable Assignment). *Let D be a non-empty* domain *of elements. A first order structure $M = (D, I, s)$ consists of*

1. *an* interpretation *I which assigns each*
   - *$T \in \mathcal{T}$ a non-empty domain $D^T \subseteq D$ s.t. $\forall S \in \mathcal{T}.\ S \preceq T \rightarrow D^S \subseteq D^T$*
   - *$f : T_1 \times \ldots \times T_n \rightarrow T \in \mathcal{F}$ a function $I(f) : D^{T_1} \times \ldots \times D^{T_n} \rightarrow D^T$*
   - *$p : T_1 \times \ldots \times T_n \in \mathcal{P}$ a relation $I(p) \subseteq D^{T_1} \times \ldots \times D^{T_n}$*
2. *a state $s : \mathcal{PV} \rightarrow D$ assigning each program variable $\mathtt{v} \in \mathcal{PV}$ of type T a value $s(\mathtt{v}) \in D^T$. We denote the set of all states by States.*

We fix the interpretation of some types and symbols: $I(\texttt{int}) = \mathbb{Z}$, $I(\top) = D$ *and the arithmetic operations* $+, -, /, \%, \ldots$ *as well as the comparators* $<, >, \leq, \geq, \doteq$ *are interpreted according to their standard semantics.*

In addition we need the notion of a variable assignment $\beta : \mathcal{V} \rightarrow D$ *which assigns each logical variable to an element of its domain.*

Definition 3 (Evaluation). *Given a first order structure* (D, I, s) *and a variable assignment* β*, we evaluate terms* t *(of type* T*) to a value* $val_{D,I,s,\beta}(t) \in D^T$*, formulas* φ *to a truth value* $val_{D,I,s,\beta}(\varphi) \in \{tt, ff\}$*, updates* $\mathcal{U}$ *to a function* $val_{D,I,s,\beta}(\mathcal{U}) : \mathcal{S} \rightarrow \mathcal{S}$*, and programs* $\texttt{p}$ *to a set of states* $val_{D,I,s,\beta}(\texttt{p}) \in 2^{\mathcal{S}}$ *with* $val_{D,I,s,\beta}(\texttt{p})$ *being either empty or a singleton set.*

A formula φ *is called* valid *iff* $val_{D,I,s,\beta}(\varphi) = tt$ *for* all *non-empty domains* D*,* all *interpretations* I*,* all *states* s *and* all *variable assignments* β*.*

The evaluation of terms and formulas without programs and updates is almost identical to standard first-order logic and omitted for brevity. The evaluation of an elementary *update* with respect to a first order structure (D, I, s) and variable assignment β is defined as follows:

$$val_{D,I,s,\beta}(\texttt{x} := t)(s') = \begin{cases} s'(\texttt{y}), & \texttt{y} \neq \texttt{x} \\ val_{D,I,s,\beta}(t), & otherwise \end{cases}$$

The evaluation of a parallel update $val_{D,I,s,\beta}(\texttt{x}_1 := t_1 \parallel \texttt{x}_2 := t_2)$ maps a state s' to a state s'' such that s'' coincides with s' except for the program variables $\texttt{x}_1, \texttt{x}_2$ which are assigned the values of the terms t_i in parallel. In case of a clash between two sub-updates (i.e., when $\texttt{x}_i = \texttt{x}_j$ for $i \neq j$), the rightmost update "wins" and overwrites the effect of the other. The meaning of a term $\{\mathcal{U}\}t$ and of a formula $\{\mathcal{U}\}\varphi$ is that the result state of the update $\mathcal{U}$ should be used for evaluating t and φ, respectively.

A *program* is evaluated to the set of states that it may terminate in when started in s. We only consider deterministic programs, so this set is always either empty (if the program does not terminate) or it consists of exactly one state.[1] The semantics of a program formula $[\texttt{p}]\varphi$ is that φ should hold in all result states of the program $\texttt{p}$, which corresponds to partial correctness of $\texttt{p}$ relative to φ.

Heap Model. The only heap objects we support in our programs (for this paper—implemented are all Java reference types) are integer typed arrays. We use an explicit heap model similar to [3]. Heaps are modelled as elements of type $\texttt{Heap}$, with two functions $store : \texttt{Heap} \times \texttt{int}[] \times \texttt{int} \times \texttt{int} \rightarrow \texttt{Heap}$ to store values on the heap and $select : \texttt{Heap} \times \texttt{int}[] \times \texttt{int} \rightarrow \texttt{int}$ to retrieve values from the heap.

For instance, $store(h, \texttt{a}, \texttt{i}, 3)$ returns a new heap which is identical to heap h except for the $\texttt{i}$-th element of array $\texttt{a}$ which is assigned the value 3. To retrieve the value of an array element `b[j]` we write $select(h, \texttt{b}, \texttt{j})$. There is a special

[1] While programs themselves are deterministic, we can introduce at least some non-determinism through the symbolic input values, which while having a single value in each model leave open which model is under consideration.

program variable `heap` which refers to the heap accessed by programs. We abbreviate $select(\mathtt{heap}, a, i)$ with $a[i]$. To ease quantification over array indices, we use $\forall x \in [l..r).\phi$ as abbreviation for $\forall x.((l \leq x \wedge x < r) \rightarrow \phi))$. Further, we write $\forall x \in arr.\phi$ for $\forall x \in [0..arr.\mathtt{length}).\phi$, where $arr.\mathtt{length}$ denotes how many elements the array arr contains.

Closely related to heaps are location sets which are defined as terms of type `LocSet`. Semantically, an element of `LocSet` describes a set of program locations. A program location is a pair (a, i) with $val_{D,I,s,\beta}(a) \in D^{\mathtt{int[]}}, val_{D,I,s,\beta}(i) \in \mathbb{Z}$ which represents the memory location of the array element $a[i]$. Syntactically, location sets can be constructed by functions over the usual set operations. We use some convenience functions and write $a[l..r]$ to represent syntactically the locations of the array elements $a[l]$ (inclusive) to $a[r]$ (exclusive). Further, we write $a[*]$ for $a[0..a.\mathtt{length}]$.

Calculus. We use a *sequent calculus* to prove that a formula is valid. *Sequents* are tuples $\Gamma \Rightarrow \Delta$ with Γ (the *antecedent*) and Δ (the *succedent*) being finite sets of formulas. The meaning $val_{D,I,s,\beta}(\Gamma \Rightarrow \Delta)$ of a sequent is the same as that of the formula $val_{D,I,s,\beta}(\bigwedge \Gamma \text{ -> } \bigvee \Delta)$. A sequent calculus *rule* is given by the rule schema,

$$\frac{seq_1 \ \ldots \ seq_n}{seq}$$

where $seq_1, \ldots, seq_n$ (the *premisses* of the rule) and seq (the *conclusion* of the rule) are sequents. A rule is *sound* iff the conclusion's validity follows from the validity of all premisses.

A sequent proof is a tree where each node is annotated with a sequent. The root node is annotated with the sequent to be proven valid. A rule is applied by matching its conclusion with a sequent of a leaf node and attaching the premisses as its children. If a branch of the tree ends in a leaf that is trivially true, the branch is called closed. A proof is closed if all its leaves are closed.

All first-order calculus rules are standard, so we explain only selected sequent calculus rules which deal with formulas involving programs. Given a suitable strategy for rule selection, the sequent calculus implements a symbolic interpreter. For example, the assignment rule for a program variable is as follows:

$$\mathsf{assignment}\quad \frac{\Gamma \Rightarrow \{\mathcal{U}\}\{\mathtt{x} := t\}[\mathtt{r}]\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{x\ =\ }t\mathtt{;\ r}]\varphi, \Delta}$$

The assignment rule for an array location adds constraints to the value the index can have, as if this value were not within the valid range for the array, an `ArrayIndexOutOfBoundsException` would be thrown, in which case we have nothing more to prove, as φ need only be shown for programs terminating without throwing exceptions.

$$\mathsf{assignment}_{array}\quad \frac{\Gamma, i \geq 0, i < \mathtt{a.length} \Rightarrow \{\mathcal{U}\}\{\mathtt{heap} := store(\mathtt{heap}, \mathtt{a}, i, t)\}[\mathtt{r}]\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{a[}i\mathtt{]\ =\ }t\mathtt{;\ r}]\varphi, \Delta}$$

The assignment rules move an assignment into an update. Updates accumulate in front of modalities during symbolic execution of the program. Once the program has been symbolically executed, the update is applied to the formula behind the modality, thereby computing its weakest precondition. Symbolic execution of *conditional statements* split the proof into two branches:

$$\mathsf{ifElse}\ \frac{\Gamma, \{\mathcal{U}\}g \Rightarrow \{\mathcal{U}\}[\mathtt{p1;\ r}]\varphi, \Delta \qquad \Gamma, \{\mathcal{U}\}\,!\,g \Rightarrow \{\mathcal{U}\}[\mathtt{p2;\ r}]\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{if\ (}g\mathtt{)\ \{p1\}\ else\ \{p2\};\ r}]\varphi, \Delta}$$

For a *loop*, the simplest approach is to *unwind* it. However, loop unwinding works only if the number of loop iterations has a concrete bound.

$$\mathsf{loopUnwind}\ \frac{\Gamma, \{\mathcal{U}\}g \Rightarrow \{\mathcal{U}\}[\mathtt{p;\ while\ (}g\mathtt{)\ \{p\};\ r}]\varphi, \Delta \qquad \Gamma, \{\mathcal{U}\}\,!\,g \Rightarrow \{\mathcal{U}\}[\mathtt{r}]\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ r}]\varphi, \Delta}$$

For unbounded loops we can use, for example, a *loop invariant* rule. To apply the loop invariant rule a loop specification consisting of a formula (the loop invariant) *Inv* and an assignable (modifies) clause *mod* is needed. The first premiss (initial case) ensures that the loop invariant *Inv* is valid before entering the loop. The second premiss (preserves case) ensures that *Inv* is preserved by an arbitrary loop iteration, while for the third premiss (use case), we have to show that after executing the remaining program, the desired postcondition φ holds.

$$\mathsf{loopInvariant}\ \frac{\begin{array}{ll} \Gamma \Rightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{initial} \\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(g \land \mathit{Inv}) \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\mathtt{p}]\mathit{Inv}, \Delta & \text{preserves} \\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\neg g \land \mathit{Inv}) \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\mathtt{r}]\varphi, \Delta & \text{use case} \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ r}]\varphi, \Delta}$$

In contrast to standard loop invariants, we keep the context (Γ, Δ) in the second and third premiss, following [2]. This is sound, because we use an *anonymizing update* $\mathcal{V}_{mod} = (\mathcal{V}^{vars}_{mod} \parallel \mathcal{V}^{heap}_{mod})$ which is constructed as follows: Let $\mathtt{x}_1, \ldots, \mathtt{x}_m$ be the program variables and $\mathtt{a}_1[t_1], \ldots, \mathtt{a}_n[t_n]$ be the array locations occurring on the left-hand sides of assignments in the loop body $\mathtt{p}$. For each $i \in \{1..n\}$ let $l_i, r_i : \mathtt{int}$ be chosen such that $val_{D,I,s,\beta}(t_i)$ at the program point $a_i[t_i] = t;$ is always between $val_{D,I,s,\beta}(l_i)$ (inclusive) and $val_{D,I,s,\beta}(r_i)$ (exclusive). Then $\mathtt{a}_i[l_i..r_i]$ are terms of type $\mathtt{LocSet}$ describing all array locations of $\mathtt{a}_i$ which might be changed by the loop. The anonymizing updates are:

$$\mathcal{V}^{vars}_{mod} := \{\mathtt{x}_1 := c_1 \parallel \ \ldots\ \parallel \mathtt{x}_m :=\ c_m\}$$
$$\mathcal{V}^{heap}_{mod} := \{\mathtt{heap} := \mathit{anon}(\ldots \mathit{anon}(\mathtt{heap}, \mathtt{a}_1[l_1..r_1], \mathit{anonH_1}), \ldots, \mathtt{a}_n[l_n..r_n],\ \mathit{anonH_n})\}$$

where the c_i are fresh constants of the same type as $\mathtt{x}_i$ and $\mathit{anonH_i}$ are fresh constants of type $\mathtt{Heap}$. The function $\mathit{anon}(h1, \mathit{locset}, h2)$ takes two heaps $h1, h2$ and a location set *locset* and returns a heap that is equal to $h1$ except for the locations mentioned in *locset* whose values are set to the values of these locations in $h2$. Informally, the anonymizing updates assign all program variables that might be changed by p and all locations enumerated in *mod* an unknown value about which only the information provided by the invariant *Inv* is available.

Updates can be simplified and applied to terms and formulas using the set of (schematic) rewrite rules given in [2,4].

2.2 Integrating Abstraction

We summarize from [1] how to integrate abstraction into our program logic. This integration provides the technical foundation for generating loop invariants.

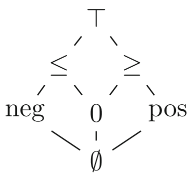
Definition 4 (Abstract Domain). *Let D be a* concrete domain *(e.g., of a first-order structure). An* abstract domain *A is a countable lattice with partial order $\sqsubseteq$ and join operator $\sqcup$ and without infinite ascending chains.*[2] *It is connected to D with an* abstraction function *$\alpha : 2^D \rightarrow A$ and a* concretization function *$\gamma : A \rightarrow 2^D$ which form a Galois connection [5].*

Instead of extending our program logic by abstract elements, we use a different approach to refer to the element of an abstract domain:

Definition 5 ($\gamma_{\alpha,\mathbb{N}}$-symbols). *Given an abstract domain $A = \{\alpha_1, \alpha_2, \ldots\}$. For each abstract element $\alpha_i \in A$ there are infinitely many constant symbols $\gamma_{\alpha_i,j} \in \mathcal{F}$, $j \in \mathbb{N}$ with $I(\gamma_{\alpha_i,j}) \in \gamma(\alpha_i)$, as well as a unary predicate χ_{α_i} where $I(\chi_{\alpha_i})$ is the characteristic predicate of set $\gamma(\alpha_i)$.*

In the definition above the interpretation I of a symbol $\gamma_{\alpha_i,j}$ is restricted to one of the concrete domain elements represented by α_i, but it is not fixed. This is important for the following notion of weakening: with respect to the symbols occurring in a given (partial) proof P and a set of formulas C, we call an update $\mathcal{U}'$ (P,C)-weaker than an update $\mathcal{U}$ if $\mathcal{U}'$ describes at least all state transitions that are also allowed by $\mathcal{U}$. Formally, given a fixed D, then $\mathcal{U}$ is weaker than $\mathcal{U}'$ iff for any first order structure $M = (D, I, s, \beta)$ there is a first order structure $M' = (D, I', s, \beta)$ with I and I' being two interpretations coinciding on all symbols used so far in P and in C and if for both structures $val_M(C) = tt$ and $val_{M'}(C) = tt$ holds, then for all program variables v the equation $val_M(\{\mathcal{U}\}v) = val_{M'}(\{\mathcal{U}'\}v)$ must hold.

Example 1. Consider the abstract sign domain for integers:



$$\gamma(\top) = \mathbb{Z} \qquad \gamma(\leq) = \{i \in \mathbb{Z} \mid i \leq 0\}$$
$$\gamma(\geq) = \{i \in \mathbb{Z} \mid i \geq 0\} \qquad \gamma(\text{neg}) = \{i \in \mathbb{Z} \mid i < 0\}$$
$$\gamma(\text{pos}) = \{i \in \mathbb{Z} \mid i > 0\} \qquad \gamma(0) = \{0\}$$
$$\gamma(\emptyset) = \{\}$$

[2] The limitation to only finite ascending chains ensures termination of our approach without the need to introduce widening operators. An extension to infinite chains with widening would be easily realizable, but so far was unnecessary.

Let P be a partial sequent proof with $\gamma_{\leq,3}$ not occurring in P. Then update $\mathtt{i} := \gamma_{\leq,3}$ is $(P, \emptyset)$-weaker than update $\mathtt{i} := -5$ or update $\mathtt{i} := c$ with a constant c (occurring in P) provided $\chi_{\leq}(c)$ holds.

The `weakenUpdate` rule from [1] integrates abstraction into our calculus:

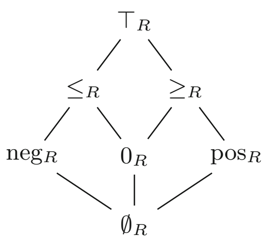
$$\mathsf{weakenUpdate}\ \frac{\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Rightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \qquad \Gamma \Rightarrow \{\mathcal{U}'\}\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}\varphi, \Delta}$$

where $\bar{\mathtt{x}}$ are all program variables occurring as left-hand sides in $\mathcal{U}$ and $\bar{\mathtt{c}}$ are fresh skolem constants. The formula $\exists\bar{\gamma}.\psi$ is a shortcut for $\exists\bar{y}.(\chi_{\bar{a}}(\bar{y}) \wedge \psi[\bar{\gamma}/\bar{y}])$, where $\bar{y} = (y_1, \ldots, y_m)$ is a list of fresh first order variables of the same length as $\bar{\gamma}$, and where $\psi[\bar{\gamma}/\bar{y}]$ stands for the formula obtained from ψ by replacing all occurrences of a symbol in $\bar{\gamma}$ with its counterpart in $\bar{y}$. Performing value-based abstraction thus becomes replacement of an update by a weaker update. In particular, we do not perform abstraction on the program, but on the *symbolic state*.

3 Loop Invariant Generation for Arrays

We refine the value-based abstraction approach from the previous section for dealing with arrays. Rather than introducing a dedicated abstract domain for arrays (e.g., abstracting an array to its length), we extend the abstract domain of the array elements to a range within the array. Given an index set (range) R, an abstract domain A for array elements can be extended to an abstract domain A_R for arrays by copying the structure of A and renaming each α_i to $\alpha_{R,i}$. The $\alpha_{R,i}$ are such that $\gamma_{\alpha_{R,i},j} \in \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid \forall k \in R.\chi_{\alpha_i}(arr[k])\}$.

Example 2. Extending the sign domain for integers gives for each range $R \subseteq \mathbb{N}$:



$$\begin{aligned}
\gamma(\top_R) &= \mathcal{D}^{\mathtt{int[]}} \\
\gamma(\leq_R) &= \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid \forall k \in R.\ arr[k] \leq 0\} \\
\gamma(\geq_R) &= \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid \forall k \in R.\ arr[k] \geq 0\} \\
\gamma(\mathrm{neg}_R) &= \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid \forall k \in R.\ arr[k] < 0\} \\
\gamma(\mathrm{pos}_R) &= \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid \forall k \in R.\ arr[k] > 0\} \\
\gamma(0_R) &= \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid \forall k \in R.\ arr[k] \doteq 0\} \\
\gamma(\emptyset_R) &= \{\}
\end{aligned}$$

Fixing $R = \{0, 2\}$, we have $\gamma(\geq_{\{0,2\}}) = \{arr \in \mathcal{D}^{\mathtt{int[]}} \mid arr[0] \geq 0 \wedge arr[2] \geq 0\}$. Importantly, the array length itself is irrelevant, provided $arr[0]$ and $arr[2]$ have the required values. Therefore the arrays (we deviate from Java's array literal syntax for clarity) $[0, 3, 6, 9]$ and $[5, -5, 0]$ are both elements of $\gamma(\geq_{\{0,2\}})$.

Of particular interest are the ranges containing (at least) all elements modified within a loop. One such range is $[0..arr.\mathtt{length})$. This range can always be taken as a fallback option if no more precise range can be found.

3.1 Loop Invariant Rule with Value and Array Abstraction

We present the rule invariantUpdate, which splits the loop invariant of the rule loopInvariant into an abstract update $\mathcal{U}'$ and an invariant *Inv*. While $\mathcal{U}'$ abstracts only the non-heap values, *Inv* can contain invariants about arrays on the heap.

invariantUpdate

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Rightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \mathtt{old} \doteq \{\mathcal{U}\}\mathtt{heap} \Rightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta \\ \Gamma, \mathtt{old} \doteq \{\mathcal{U}\}\mathtt{heap},\ \{\mathcal{U}'_{mod}\}(g \wedge \mathit{Inv}),\ \{\mathcal{U}'_{mod}\}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c}) \Rightarrow \exists\bar{\gamma}.\{\mathcal{U}'_{mod}\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \mathtt{old} \doteq \{\mathcal{U}\}\mathtt{heap},\ \{\mathcal{U}'_{mod}\}(g \wedge \mathit{Inv}) \Rightarrow \{\mathcal{U}'_{mod}\}[\mathtt{p}]\mathit{Inv}, \Delta \\ \Gamma, \mathtt{old} \doteq \{\mathcal{U}\}\mathtt{heap},\ \{\mathcal{U}'_{mod}\}(\neg g \wedge \mathit{Inv}) \Rightarrow \{\mathcal{U}'_{mod}\}[r]\varphi, \Delta \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{while}\ (g)\ \{p\};\ r]\varphi, \Delta}$$

The first premiss is identical to the left premiss of weakenUpdate, introducing a suitable abstraction $\mathcal{U}'$ of $\mathcal{U}$. The symbols $\bar{\mathtt{x}}, \bar{c}, \bar{\gamma}$ and $\exists\bar{\gamma}\varphi$ are also defined as in the weakenUpdate rule. From $\mathcal{U}'$ we obtain $\mathcal{U}'_{mod} := (\mathcal{U}' \parallel \mathcal{V}^{heap}_{mod})$ by anonymizing the heap locations that might be changed in the loop body as explained in Sect. 2.1. Anonymization of local variables $\mathcal{V}^{vars}_{mod}$ is not required, as it is already part of $\mathcal{U}'$. More precisely, $\mathcal{U}'$ can contain updates $\mathtt{x} := \gamma_{\alpha_i,j}$ which combine the anonymization of $\mathcal{V}^{vars}_{mod}$ with an invariant based on the abstract domain.

The identifier `old` is a fresh constant and used in the invariant *Inv* to refer to the heap before loop execution. *Inv* contains invariants related to the heap. Intuitively $\mathcal{U}'_{mod}$ and *Inv* together express all states in which the program could be before or after any iteration of the loop. The first two premisses together ensure that the abstract update $\mathcal{U}'_{mod}$ and the invariant *Inv* are a valid weakening of the original update $\mathcal{U}$. The following two premisses ensure that $\mathcal{U}'_{mod}$ and *Inv* actually constitute a loop invariant: for any given interpretation of $\mathcal{U}'_{mod}$ satisfying *Inv* executing the loop body results in an abstract state no weaker than $\mathcal{U}'_{mod}$ in which *Inv* remains valid. The last premiss is the use case, where the desired postcondition φ must be established based on the state after exiting the loop and after execution of the remaining program.

Listing 1.1. Example

```
i = 0; j = 0;
while(i < a.length) {
  if (a[j] > 0) j++;
  b[i] = j;
  c[2*i] = 0;
  i++;
}
```

Given the program `p` in Listing 1.1, we can apply the assignment rule twice to $\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{p}]\varphi, \Delta$ which leads to $\Gamma \Rightarrow \{\mathcal{U} \parallel \mathtt{i} := 0 \parallel \mathtt{j} := 0\}[\mathtt{while}\ldots]\varphi, \Delta$. Now invariantUpdate can be applied with the values in Fig. 1: the update $\mathcal{U}'$ is equal to the original update $\mathcal{U}$ except for the values of `i` and `j` which can both be any non-negative number. The arrays `b` and `c` have (partial) ranges anonymized, while `a` is not anonymized as it is not changed by the loop. The invariants in *Inv* express that (a) `a` contains positive values at all positions prior to the current value of `j`, (b) the anonymized values[3] in `b` are all non-negative, and (c) the anonymized values in `c` are equal to 0 or to their original values, if the loop has not (yet) modified them.

[3] Note choosing the range $[0..\mathtt{i})$ for the array `b` is sound even when $\mathtt{i} \geq \mathtt{b.length}$, as an uncaught `ArrayIndexOutOfBoundsException` is treated as non-termination.

$$\begin{aligned}
\mathcal{U}' &= (\mathcal{U} \parallel \mathtt{i} := \gamma_{\geq,1} \parallel \mathtt{j} := \gamma_{\geq,2})\\
\mathcal{V}^{heap}_{mod} &= \mathtt{heap} := anon(anon(\mathtt{heap}, \mathtt{b}[0..i], anonHeap_1),\ \mathtt{c}[*], anonHeap_2)\\
Inv &= (\forall k \in [0..\mathtt{j}).\ \chi_{>}(\mathtt{a}[k]))\\
&\quad \wedge\ (\forall k \in [0..\mathtt{i}).\ \chi_{\geq}(\mathtt{b}[k]))\\
&\quad \wedge\ (\forall m \in c.\ (m < 2 * i \wedge m\%2 \doteq 0) \rightarrow \chi_0(\mathtt{c}[2 * m]))\\
&\quad \wedge\ (\forall m \in c.\ \neg(m < 2 * i \wedge m\%2 \doteq 0) \rightarrow (\mathtt{c}[m] \doteq select(\mathtt{old}, \mathtt{c}, m)))
\end{aligned}$$

Fig. 1. Values for invariantUpdate

Algorithm 1. Generating an abstract update and invariant fixpoint

input : the sequent *seq*
output: the fixpoint $\mathcal{U}'$ with valid $\mathcal{V}^{heap}_{mod}$ and *Inv*, as $(\mathcal{U}'_m, Inv)$

$\mathcal{U}'_m \leftarrow \mathcal{U}$;
while *true* **do**
  `/*` *seq* `is of the form:` $\Gamma \Rightarrow \{\mathcal{U}'_m\}[\mathtt{while\ (}g\mathtt{)\ \{}p\mathtt{\};\ }r]\varphi, \Delta$ `*/`
  $\mathcal{U}^* \leftarrow \mathcal{U}'_m$; $Inv \leftarrow \Gamma \cup !\Delta$;
  $seq \leftarrow\ (\Gamma, \{\mathcal{U}'_m\}g \Rightarrow \{\mathcal{U}'_m\}[p\mathtt{;while(}g\mathtt{)\{}p\mathtt{\};}r]\varphi, \Delta)$;
  perform symbolic execution on *seq*;
  `/* all branches either closed or loop entry reached again */`
  foreach *open branch with* $\Gamma_i \Rightarrow \{\mathcal{U}_i\}[\mathit{while\ (g)\ \{p\};\ r}]\varphi, \Delta_i$ **do**
    $(Inv, \mathcal{U}^*) \leftarrow (Inv, \mathcal{U}^*)\ \dot{\sqcup}\ (\Gamma_i \cup !\Delta_i, \mathcal{U}_i)$; `// see Definition 6 for` $\dot{\sqcup}$
  end
  if $\mathcal{U}'_m$ *is (P,Inv)-weaker than* $\mathcal{U}^*$ **then**
    return $(\mathcal{U}'_m, Inv)$;
  end
  $\mathcal{U}'_m \leftarrow \mathcal{U}^*$; $\Gamma \leftarrow \Gamma \cup \{\mathcal{U}'_m\}Inv$;
  $seq \leftarrow (\Gamma \Rightarrow \{\mathcal{U}'_m\}[\mathtt{while\ (}g\mathtt{)\ \{}p\mathtt{\};\ }r]\varphi, \Delta)$;
end

3.2 Computation of the Abstract Update and Invariants

We generate the values of $\mathcal{U}'$, $\mathcal{V}^{heap}_{mod}$ and *Inv* as required by invariantUpdate *automatically* in a side proof, by symbolic execution of single loop iterations until a fixpoint is found. For each value change of a variable during the execution of a loop iteration the abstract update $\mathcal{U}'$ will set this variable to a value at least as weak as its value both before and after loop execution. We generate $\mathcal{V}^{heap}_{mod}$ and *Inv* by examining each array modification[4] and anonymizing the entire range within the array (expressed in $\mathcal{V}^{heap}_{mod}$) while adding a partial invariant to the set *Inv*. Once a fixpoint for $\mathcal{U}'$ is reached, we can refine $\mathcal{V}^{heap}_{mod}$ and *Inv* by performing in essence a second fixpoint iteration, this time anonymizing possibly smaller ranges and potentially adding more invariants. We explain this now step by step.

[4] Later we also examine each array access (read or write) in `if`-conditions to gain invariants such as $\forall k \in [0..\mathtt{j}).\ \chi_{>}(select(\mathtt{heap}, \mathtt{a}, k))$ in the example above.

Algorithm 2. Concrete update join $\dot{\sqcup}_{upd}$

input : $((C_1,\mathcal{U}_1),(C_2,\mathcal{U}_2))$
output: the weaker constraint/update pair $(C_{res},\mathcal{U}_{res})$

$(\mathcal{U}_{res} \parallel \texttt{heap} := h') \leftarrow (C_1,\mathcal{U}_1)\ \sqcup_{abs}\ (C_2,\mathcal{U}_2)$; `// heap update` h' `ignored`
$(C_{res}, h) \leftarrow (C_1, \{\mathcal{U}_1\}\texttt{heap})\ \hat{\sqcup}\ (C_2, \{\mathcal{U}_2\}\texttt{heap})$; `// see Definition 7 for` $\hat{\sqcup}$
$\mathcal{U}_{res} \leftarrow (\mathcal{U}_{res} \parallel \texttt{heap} := h)$;
return $(C_{res},\mathcal{U}_{res})$

The first step is to generate $\mathcal{U}'$ (with valid but imprecise $\mathcal{V}^{heap}_{mod}$ and *Inv*). For this we use Algorithm 1 with input $seq = (\Gamma \Rightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{}p\texttt{\}; }r]\varphi, \Delta)$, the conclusion of invariantUpdate.

The algorithm requires to compute the join $\dot{\sqcup}$ of pairs of invariants and updates. In [1] a concrete implementation for joining updates $(C_1,\mathcal{U}_1) \sqcup_{abs} (C_2,\mathcal{U}_2)$ with

$$\sqcup_{abs} : (2^{For} \times Upd) \times (2^{For} \times Upd) \to Upd$$

was computed as follows: For each update $\texttt{x} := v$ in $\mathcal{U}_1$ or $\mathcal{U}_2$ the generated update is $\texttt{x} := v$, if $\{\mathcal{U}_1\}x \doteq \{\mathcal{U}_2\}x$ under C_1, C_2 respectively. Otherwise it is $\texttt{x} := \gamma_{\alpha_i,j}$ for some α_i where $C_1 \Rightarrow \chi_{\alpha_i}(\{\mathcal{U}_1\}x)$ and $C_2 \Rightarrow \chi_{\alpha_i}(\{\mathcal{U}_2\}x)$ are valid. For a simple heap abstraction this returns (for some $n \in \mathbb{N}$) $\texttt{heap} := \gamma_{\top,n}$ for any non-identical heaps. As we wish to join the heaps meaningfully, which leads to the generation of constraints, our update join operation has the signature

$$\dot{\sqcup} : (2^{For} \times Upd) \times (2^{For} \times Upd) \to (2^{For} \times Upd).$$

Definition 6 (Joining Updates). *Any operation $\dot{\sqcup}$ satisfying the following properties is an* update join operation*: Let $\mathcal{U}_1$, $\mathcal{U}_2$ be arbitrary updates in a proof P and let C_1, C_2 be formula sets representing constraints on the update values. Then for $(C,\mathcal{U}) = (C_1,\mathcal{U}_1)\ \dot{\sqcup}\ (C_2,\mathcal{U}_2)$ the following holds for $i \in \{1,2\}$: (a) $\mathcal{U}$ is (P, C_i)-weaker than $\mathcal{U}_i$, (b) $C_i \Rightarrow \{\mathcal{U}_i\}\bigwedge C$, and (c) $\dot{\sqcup}$ is associative and commutative up to first-order reasoning.*

Let $C_1,\mathcal{U}_1$ and $C_2,\mathcal{U}_2$ be constraint/update pairs. $(C_1,\mathcal{U}_1)\ \dot{\sqcup}_{upd}\ (C_2,\mathcal{U}_2)$ computes the update $\mathcal{U}_{res}$ and the set of heap restrictions C_{res} as shown in Algorithm 2. Intuitively, if all the restrictions in C_{res} are satisfied by the heap under update $\mathcal{U}_{res}$ then $\mathcal{U}_{res}$ is the lattice join of $\mathcal{U}_1$ and $\mathcal{U}_2$.

Lemma 1. *$\dot{\sqcup}_{upd}$ is an update join operator.*
The proof is in the appendix of the extended technical report [6].

Definition 7 (Joining Heaps). *Any operator with the signature*

$$\hat{\sqcup} : (2^{For} \times Trm_{\texttt{Heap}}) \times (2^{For} \times Trm_{\texttt{Heap}}) \to (2^{For} \times Trm_{\texttt{Heap}})$$

is a heap join operator *if it satisfies the properties: Let h_1, h_2 be arbitrary heaps in a proof P, C_1, C_2 be formula sets representing constraints on the heaps* *(and possibly also on other update values) and let $\mathcal{U}$ be an arbitrary update. Then for $(C, h) = (C_1, h_1) \hat{\sqcup} (C_2, h_2)$ the following holds for $i \in \{1, 2\}$: (a) $(\mathcal{U} \parallel \mathtt{heap} := h)$ is (P, C_i)-weaker than $(\mathcal{U} \parallel \mathtt{heap} := h_i)$, (b) $C_i \Rightarrow \{\mathcal{U} \parallel \mathtt{heap} := h_i\} \bigwedge C$, and (c) $\hat{\sqcup}$ is associative and commutative up to first-order reasoning.*

We define the set of *normal form heaps* $\mathcal{H}_{NF} \subset \mathit{Trm}_{\mathtt{Heap}}$ to be those heap terms that extend `heap` with an arbitrary number of preceding stores or anonymizations. For a heap term $h \in \mathcal{H}_{NF}$ we define

$$\mathit{writes}(h) := \begin{cases} \emptyset & \text{if } h = \mathtt{heap} \\ \{h\} \cup \mathit{writes}(h') & \text{if } h = \mathit{store}(h', a, \mathit{idx}, v) \text{ or } h = \mathit{anon}(h', a[l..r], h'') \end{cases}$$

A concrete implementation $\hat{\sqcup}_{heap}$ of $\hat{\sqcup}$ is given as follows: We reduce the signature to $\hat{\sqcup}_{heap} : (2^{For} \times \mathcal{H}_{NF}) \times (2^{For} \times \mathcal{H}_{NF}) \to (2^{For} \times \mathcal{H}_{NF})$. This ensures that all heaps we examine are based on `heap` and is a valid assumption when taking the program rules into account, as these maintain this normal form. As both heaps are in normal form, they must share a common subheap (at least `heap`). The largest common subheap of h_1, h_2 is defined as $\mathit{lcs}(h_1, h_2)$ and all writes performed on this subheap can be given as $\mathit{writes}_{lcs}(h_1, h_2) := \mathit{writes}(h_1) \cup \mathit{writes}(h_2) \setminus (\mathit{writes}(h_1) \cap \mathit{writes}(h_2))$. Algorithm 3 shows how the join of heaps $(C_1, h_1) \hat{\sqcup}_{heap} (C_2, h_2)$ is calculated.

Lemma 2. *The concrete implementation $\hat{\sqcup}_{heap}$ is a heap join operator on the reduced signature $(2^{For} \times \mathcal{H}_{NF}) \times (2^{For} \times \mathcal{H}_{NF}) \to (2^{For} \times \mathcal{H}_{NF})$.*

The proof is in the appendix of the extended technical report [6].

Example 3. With the precondition $P = \forall n \in \mathtt{b}.\ \mathit{select}(\mathtt{heap}, \mathtt{b}, n) \doteq -1$ and the program in Listing 1.1, we demonstrate the first steps of Algorithm 1 with $\mathit{seq} = P \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{j} := 0\}[\mathtt{while}\ldots]\varphi$: After initialization $\mathit{Inv} = \{P\}$ and $\mathcal{U}^* = (\mathtt{i} := 0 \parallel \mathtt{j} := 0)$. At line 8 of Algorithm 1 we have two open branches:

$$P, \{\mathcal{U}^*\}g, \neg(\mathit{select}(\mathtt{heap}, \mathtt{a}, 0) > 0) \Rightarrow$$
$$\{\mathtt{i} := 1 \parallel \mathtt{j} := 0 \parallel \mathtt{heap} := \mathit{store}(\mathit{store}(\mathtt{heap}, \mathtt{b}, 0, 0), \mathtt{c}, 0, 0)\}[\mathtt{while}\ldots]\varphi \quad (1)$$
$$P, \{\mathcal{U}^*\}g, \mathit{select}(\mathtt{heap}, \mathtt{a}, 0) > 0 \Rightarrow$$
$$\{\mathtt{i} := 1 \parallel \mathtt{j} := 1 \parallel \mathtt{heap} := \mathit{store}(\mathit{store}(\mathtt{heap}, \mathtt{b}, 0, 1), \mathtt{c}, 0, 0)\}[\mathtt{while}\ldots]\varphi \quad (2)$$

Algorithm 3. Concrete heap join $\hat{\sqcup}_{heap}$

input : $((C_1, h_1), (C_2, h_2))$
output: the weaker constraint/heap pair (C_{res}, h_{res})

$h_{res} \leftarrow \mathit{lcs}(h_1, h_2)$; $C_{res} \leftarrow \emptyset$; $W \leftarrow \mathit{writes}_{lcs}(h_1, h_2)$;
foreach $\mathit{anon}(h, a[l..r], \mathit{anonHeap})$ *or* $\mathit{store}(h, a, \mathit{idx}, v) \in W$ **do**
  $h_{res} \leftarrow \mathit{anon}(h_{res}, a[*], \mathit{anonHeap}')$;
  $i_1, i_2 \leftarrow$ the indices of the smallest α_{i_j} such that $C_j \Rightarrow \forall k \in a.\ \chi_{\alpha_{i_j}}(\mathit{select}(h_j, a, k))$;
  $C_{res} \leftarrow C_{res} \cup \{\forall k \in a.\ \chi_{\alpha_{i_1} \sqcup \alpha_{i_2}}(\mathit{select}(\mathtt{heap}, a, k))\}$
end

We can use Algorithm 2 to compute the update join of the original $(\{P\}, \mathcal{U}^*)$ with $(\{P, \{\mathcal{U}^*\}g, \neg(\mathit{select}(\mathtt{heap}, \mathtt{a}, 0) > 0)\}, \mathtt{i} := 1 \parallel \mathtt{j} := 0 \parallel \mathtt{heap} := h_1)$ provided by (1), where $h_1 = \mathit{store}(\mathit{store}(\mathtt{heap}, \mathtt{b}, 0, 0), \mathtt{c}, 0, 0)$. This produces $(C_{res}, \mathtt{i} := \gamma_{\geq,1} \parallel \mathtt{j} := 0 \parallel \mathtt{heap} := h_{res})$, where (C_{res}, h_{res}) is a heap join of $(\{P\}, \mathtt{heap})$ and $(\{P, \{\mathcal{U}^*\}g, \neg(\mathit{select}(\mathtt{heap}, \mathtt{a}, 0) > 0)\}, h_1)$. Algorithm 3 can compute the latter as follows: the largest common subheap is $h' = \mathtt{heap}$, so we have $W = \{\mathit{store}(\mathit{store}(\mathtt{heap}, \mathtt{b}, 0, 0), \mathtt{c}, 0, 0), \mathit{store}(\mathtt{heap}, \mathtt{b}, 0, 0)\}$, therefore:

$$C_{res} = \{\forall m \in \mathtt{b}.\ \chi_{\leq}(\mathit{select}(\mathtt{heap}, \mathtt{b}, m)),\ \forall n \in \mathtt{c}.\ \chi_{\top}(\mathit{select}(\mathtt{heap}, \mathtt{c}, n))\}$$
$$h_{res} = \mathit{anon}(\mathit{anon}(\mathtt{heap}, \mathtt{b}[*], \mathit{anonH}_1), \mathtt{c}[*], \mathit{anonH}_2)$$

At line 9 of Algorithm 1 we have $\mathcal{U}^* = (\mathtt{i} := \gamma_{\geq,1} \parallel \mathtt{j} := 0 \parallel \mathtt{heap} := h_{res})$ and $\mathit{Inv} = C_{res}$. Now the algorithm joins updates with the second open branch, checks if a fixpoint has been found (it has not) and enters the next iteration.

4 Symbolic Pivots

Algorithm 1 computes an abstract update $\mathcal{U}'$ expressing the state of all non-heap program variables before and after each loop iteration and, in particular, before entering the loop. It also computes $\mathcal{V}^{heap}_{mod}$ and *Inv*, which give information about the state of the heap before and after each loop iteration. However, as a consequence of the definition of heap joins in Algorithm 3, this information is rather weak as it assumes any update to an array element could cause a change at any index. To remedy this situation we refine $\mathcal{U}'$. The main idea is to keep track of the ranges within a given array where a modification has been made and where it has not been modified. The boundary indices of such ranges are often called *pivot* in array algorithms. To obtain invariants that are valid in any state before and after a loop iteration, obviously, these pivots must be symbolic.

Listing 1.2. Inferring Modified Array Elements

```
int i = 0;
int j = 5;
while (j < a.length) {
  i = j + 1;
  d[i] = j;
  j = i + 1;
  i = 0;
}
```

We start with an example that illustrates the difficulties of computing *symbolic pivots*. Consider Listing 1.2. A naïve approach to recording pivots would be to consider just the array modification statement, here "`d[i] = j;`", and infer that the modifications to `d` occur at the index given by the value of `i`. But this is completely wrong here. Variable `i` has the constant value `0` at the beginning of each iteration, while the array modifications occur at indices based on the value of `j`. This is problematic to detect for analyses based on control flow graphs, but easy for our value-sensitive approach. The reason is that the update created during a loop iteration of the example immediately shows that the value of `i` is unchanged.

The problem remains that while we know, for example, that in the first iteration of the loop the array element `d[6]` is set to `5`, we cannot infer *why* that particular index was chosen. But we need to know this to generate valid invariants. McMillan [7] points out this problem while analyzing multiple, successive iterations of a loop and then attempts to infer why array elements at specific indices were modified. Our approach allows a more uniform analysis: first we calculate an over-approximation of the modified ranges, resulting in γ-terms which are integer abstractions that constitute correct boundaries of array ranges. Based on these γ-terms we then execute symbolically one iteration of the loop whereby we keep track of modifications to array elements.

Example 4. Running Algorithm 1 on the loop in Listing 1.2 results in the following updates for non-heap variables: $(\mathtt{i} := 0 \parallel \mathtt{j} := \gamma_{>,1})$ Symbolic execution of a loop iteration started with $\gamma_{>,1}$ as the value of $\mathtt{j}$ leads to the following update:

$$\mathtt{i} := 0 \parallel \mathtt{j} := \gamma_{>,1} + 2 \parallel \mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{d}, \gamma_{>,1} + 1, \gamma_{>,1})$$

Value $\gamma_{>,1}$ was the initial value of $\mathtt{j}$, so we can conclude that the array elements are modified at the value of $\mathtt{j} + 1$ in each iteration.

Now we describe how this can be made to work in the general case. Consider the sequent $\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{while}(g)\{p\};\ r]\varphi, \Delta$ and the update $\mathcal{U}'$ computed by Algorithm 1. Then an update $\mathcal{U}''$ which maps all variables but $\mathtt{heap}$ just as $\mathcal{U}'$ does and maps $\mathtt{heap}$ as the original $\mathcal{U}$ did remains weaker than $\mathcal{U}$, as $\mathcal{U}'$ is weaker than $\mathcal{U}$. Applying Algorithm 1 to sequent $\Gamma \Rightarrow \mathcal{U}''[\mathtt{while}(g)\{p\};\ r]\varphi, \Delta$ we obtain open subgoals of the form $\Gamma_i \Rightarrow \{\mathcal{U}_i\}[\mathtt{while}\ (g)\ \{p\};\ r]\varphi, \Delta_i$. Aside from the values for $\mathtt{heap}$, $\mathcal{U}'$ is weaker than $\mathcal{U}_i$, as $\mathcal{U}'$ is a fixpoint. We therefore do not have to join any non-heap variables when computing $(\mathcal{U}^*, \mathit{Inv})$, as fixpoints for those are already calculated and will not change.

When joining constraint/heap pairs we distinguish between three types of writes (see Sect. 3.2): (a) anonymizations, which are kept, as well as any invariants generated for them occurring in the constraints, (b) stores to concrete indices, for which we create a store to the index either of the explicit value (if equal in both heaps) or of a fresh $\gamma_{i,j}$ of appropriate type, and (c) stores to variable indices, which we turn into symbolic pivots (and, hence, stronger invariants) as follows. Given a $\mathit{store}(h, a, \mathit{idx}, v)$ to a variable index, idx is expressible as a function $\mathit{index}(\gamma_{i_0,j_0}, \ldots, \gamma_{i_n,j_n})$. These γ_{i_x,j_x} can be linked to program variables in the update $\mathcal{U}'$, which contains updates $\mathtt{pv}_x := \gamma_{i_x,j_x}$. We can therefore represent idx as a function $\mathit{sp}(\ldots \mathtt{pv}_x \ldots)$ and call it a *symbolic pivot*.

Example 5. Continuing Example 4, $\mathtt{d}$ is modified at index $\mathit{index}(\gamma_{>,1}) = \gamma_{>,1}+1$. As $\gamma_{>,1}$ was the value of $\mathtt{j}$, the symbolic pivot is $\mathit{sp}(\mathtt{j}) = \mathtt{j} + 1$.

The final step is to exploit the shape of symbolic pivots to derive certain kinds of inductive invariants. For this we need two abbreviations. Formula $P(\mathcal{W})$ is defined for a fixed update $\mathcal{U}$, array a, and symbolic pivot sp as: $P(\mathcal{W}) := \forall k \in [\{\mathcal{U}\}\mathit{sp}..\{\mathcal{W}\}\mathit{sp}).\ \{\mathcal{W}\}\chi_{\alpha_j}(\mathit{select}(\mathtt{heap}, a, k))$. Then $P(\mathcal{U})$ is trivially valid, as we are quantifying over an empty set. Likewise, it is easy to show that the instance $Q(\mathcal{U})$ of the following formula $Q(\mathcal{W})$ is valid:

$$\forall k \notin [\{\mathcal{U}\}\mathit{sp}..\{\mathcal{W}\}\mathit{sp}).\ \mathit{select}(\{\mathcal{W}\}\mathtt{heap}, \{\mathcal{W}\}a, k) \doteq \mathit{select}(\{\mathcal{U}\}\mathtt{heap}, \{\mathcal{W}\}a, k)$$

Therefore, anonymizing an array a with $anon(h, a[*], anonHeap)$ and adding invariants $P(\mathcal{U}^*)$ and $Q(\mathcal{U}^*)$ for the contiguous range $[\{\mathcal{U}\}sp..\{\mathcal{U}^*\}sp)$ is inductively sound if $P(\mathcal{U}') \Rightarrow P(\mathcal{U}_i)$ and $Q(\mathcal{U}') \Rightarrow Q(\mathcal{U}_i)$ hold. The same is true for the range $[\{\mathcal{U}^*\}sp..\{\mathcal{U}\}sp)$, hence w.l.o.g. in the sequel $\{\mathcal{U}^*\}sp \geq \{\mathcal{U}\}sp$.

Definition 8 (Iteration Affine). *Given a sequent $\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{p}]\varphi, \Delta$ where* `p` *starts with* `while`*, a term t is called* iteration affine*, if there exists $step \in \mathbb{Z}$ such that for any $n \in \mathbb{N}$, if we unwind and symbolically execute the loop n times, for each subgoal with sequent $\Gamma_i \Rightarrow \{\mathcal{U}_i\}[\mathtt{p}]\varphi, \Delta_i$ there is some v, such that $\Gamma_i \cup !\Delta_i \Rightarrow \{\mathcal{U}_i\}t \doteq v$ and $\Gamma \cup !\Delta \Rightarrow \{\mathcal{U}\}t + n * step \doteq v$.*

From iteration affine symbolic pivots we can directly construct inductive invariants over array ranges as follows. First, after unwinding a loop body once we posit a symbolic pivot sp as iteration affine using $step := (\{\mathcal{U}'\}sp) - (\{\mathcal{U}\}sp)$, where $\mathcal{U}'$ is the program state after executing the loop body. Then simply add the constraint $n \geq 0 \wedge (\{\mathcal{U}\}sp) + n * step \doteq v$ for a fresh n in further fixpoint iterations and ensure that $(\{\mathcal{U}'\}sp) \doteq v + step$ holds. If this is not the case, then sp is not iteration affine and we remove the constraint in following fixpoint iterations. Otherwise, once a fixpoint is found we know the exact array elements that may be modified, as sp is iteration affine. To express an affine range as a location set is difficult. To avoid it, we anonymize the entire array and create the following invariants for the modified and unmodified partitions (using the symbols of Definition 8) where $M := (k \geq \{\mathcal{U}\}sp \wedge k < sp \wedge (k - \{\mathcal{U}\}sp)\%step \doteq 0)$:

$$\forall k \in arr.\ M \rightarrow \chi(arr[k]) \tag{3}$$

$$\forall k \in arr.\ \neg M \rightarrow arr[k] \doteq select(\{\mathcal{U}\}\mathtt{heap}, arr, k) \tag{4}$$

Example 6. This symbolic pivot $\mathtt{j}+1$ from Example 5 is iteration affine, expressible as $6 + it * 2$ for the it-th iteration, based on the initial value of $j + 1$ being 6 and each successive value for $j + 1$ being two more than the last value. We therefore store in variable `old` the value of `heap` before the loop, anonymize all elements of `d` and add the invariants:

$$\forall k \in \mathtt{d}.\ (k \geq 6 \wedge k < \mathtt{j} + 1 \wedge (k - 6)\%2 \doteq 0) \rightarrow \chi_{>}(\mathtt{d}[k])$$
$$\forall k \in \mathtt{d}.\ (k < 6 \vee k \geq \mathtt{j} + 1 \vee (k - 6)\%2 \neq 0) \rightarrow \mathtt{d}[k] \doteq select(\mathtt{old}, \mathtt{d}, k)$$

Besides array modifications, our approach can also add invariants based on read-only array accesses that influence control flow. The steps involved are similar: *(i)* calculate the symbolic pivot, *(ii)* determine whether it is iteration affine, and *(iii)* generate an invariant with a contiguous or affine range. However, as no anonymization takes place for an unmodified array, no invariant of the form (4) is generated.

Our approach automatically produces all invariants in Fig. 1: affine invariants for array `c` and contiguous invariants for array `b` and the unmodified array `a`.

5 Implementation

The presented approach has been implemented as a proof-of-concept (available at http://www.key-project.org/symbolic-pivots/) and integrated into a variant

Table 1. Experimental results.

Method	LocSets modified	Automatically generated array invariants
`arrayInit`	$a[0..\mathtt{i}]$	$\forall j_1 \in [0..\mathtt{i}).\ \mathtt{a}[j_1] \doteq 0)$
`arrayMax`	-	$\forall j_7 \in [0..i).\ \mathtt{a}[j_7] \leq \mathtt{max}^{\mathrm{a}}$
`arraySplit`	$b[0..\mathtt{j}]$, $c[0..\mathtt{k}]$	$\forall j_5 \in [0..\mathtt{j}).\ \mathtt{b}[j_5] > 0),\ \forall j_6 \in [0..\mathtt{k}).\ \mathtt{c}[j_6] \leq 0)$
`firstNotNull`	-	$\forall j_0 \in [0..i).\ \mathtt{a}[j_0] \doteq 0$
`sentinel`	-	$\forall j_{11} \in [0..i).\ \mathtt{a}[j_{11}] \neq \mathtt{x}$

[a] Relational abstract domains are not directly possible in our approach, but we can generate invariants containing terms such as $\chi_{\leq}(\mathtt{a}[j_7] - \mathtt{max})$, which is equivalent to the relational invariant $\mathtt{a}[j_7] \leq \mathtt{max}$.

of the KeY verification system for Java, which focuses on checking programs for secure information flow. In this context less strong invariants than for functional verification are sufficient and the precision of the automatically generated invariants is, therefore, good enough in many cases.

In addition to the array example in Listing 1.1 we created a small test suite based on benchmarks given in related work [8,9] and display the resulting array invariants produced by our approach in Table 1. The generation time is still quite high, ranging from a few seconds to ten minutes. The relatively long runtime is due to the current status of the implementation, which does not perform any caching and is instrumented with debug statements. In addition, the implementation currently uses solely the internal proof producing theorem prover for the invariant computation. Switching to an SMT solver for pure first-order steps should increase speed significantly. One additional reason for long runtimes is that in addition to the invariants generated for the array elements themselves, we also generate some useful invariants only semi-related to the array elements, such as the following for the `arraySplit` example (using Java notation for conditional terms):

$$\mathtt{i} \leq \mathtt{a.length} \quad \wedge \quad \mathtt{j} = \sum_{q=0}^{\mathtt{i}-1} (\mathtt{a}[q] > 0\ ?\ 1 : 0) \quad \wedge \quad \mathtt{k} = \sum_{q=0}^{\mathtt{i}-1} (\mathtt{a}[q] > 0\ ?\ 0 : 1).$$

6 Related Work

To find a fixpoint for non-heap variables we perform something akin to *array smashing* [10] for any array modification in a loop. Our refinements based on symbolic pivots later remedy much of the lost precision. In [11] invariants based on linear loop-dependent scalars (i.e. variables which can be modified by a loop) are computed. In [12] variables within a loop are specified according to a number of properties: increasing, dense, etc. There are similarities between iteration affine variables and linear loop-dependent scalars as well as the variables determined in [12]. Our approach uses symbolic execution to determine iteration affine *terms*, in particular in array indices, which do not have to coincide with iteration affine variables. *Range predicates* are used in [13] to express knowledge about all elements of an array within a given range. These could be used to express our affine range invariants about modified elements, however they are not strong enough to express the affine range invariants about unmodified elements. In [14] abstract domains need to be explicitly supplied for the array indices, offering more possibilities than our approach. However, our notion of iteration affine indices offers the equivalent of an infinite number of abstract domains for array indices which do not need to be explicitly supplied. Their approach also does not allow for additional information to be added about array elements without overwriting old information. In contrast to CEGAR [15] which starts abstract and refines the abstraction stepwise, we start with a fully precise modeling and perform abstraction only on demand and confined to a part of the state. In [16] arrays are modeled as (many) contiguous partitions, while we allow both contiguous partitions as well as affine ranges. In [8] templates are used to introduce quantified formulas from quantifier-free elements, while we allow the underlying abstract domain to function as a "template." In [9] modification of array elements is modeled by abstracting the program: the array is replaced by multiple array slices containing abstract values. The text of the program is used to influence which slices are generated. By abstracting only program states, we can keep much higher precision. Further, our use of symbolic execution lets us view the *result* of the loop body, rather than just the text, allowing two equivalent loop bodies to be treated the same with our approach. In [17] *foot-prints* are introduced which track what part of the program state can be changed by a statement. Using these they can reason about recursive programs containing unbounded arrays (modelled as total functions).

7 Conclusion and Future Work

We presented a novel approach to generate loop invariants for loops that perform operations on arrays. It integrates smoothly into a framework which combines deduction and abstract interpretation. As future work we intend to improve the flexibility of the partitioning by supporting more shapes than affine ranges and on improvements needed for the treatment of nested loops. We will also extend our approach to the diamond modality $\langle\cdot\rangle\cdot$, which expresses total correctness. We investigate several speed ups including avoidance of repeated symbolic execution by reusing the symbolic execution tree of one general run, cache strategies for joins and use of an SMT solver for pure first-order reasoning steps. We intend to integrate our approach into the framework presented in [18] to avoid their need for user specified loop invariants.

References

1. Bubel, R., Hähnle, R., Weiß, B.: Abstract interpretation of symbolic execution with explicit state updates. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 247–277. Springer, Heidelberg (2009)

2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Weiß, B.: Deductive Verification of Object-Oriented Software – Dynamic Frames, Dynamic Logic and Predicate Abstraction. Ph.D. thesis, KIT., January 2011
4. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th Symposium on Principles of Programming Languages (POPL), pp. 238–252. ACM (1977)
6. Wasser, N., Bubel, R., Hähnle, R.: TR: array abstraction with symbolic pivots. Technical report, Department of Computer Science, Technische Universität Darmstadt, Germany, August 2015
7. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
8. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. SIGPLAN Not. **43**(1), 235–246 (2008)
9. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. SIGPLAN Not. **43**(6), 339–348 (2008)
10. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg (2002)
11. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
12. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
13. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
14. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th Symposium on Principles of Programming Languages, POPL 2011, pp. 105–118. ACM (2011)
15. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
16. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. SIGPLAN Not. **40**(1), 338–350 (2005)
17. de Boer, F.S., de Gouw, S.: Being and change: reasoning about invariance. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Olderog-Festschrift. LNCS, vol. 9360, pp. 191–204. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23506-6_13
18. Hentschel, M., Käsdorf, S., Hähnle, R., Bubel, R.: An interactive verification tool meets an IDE. In: Proceedings of the 11th International Conference on Integrated Formal Methods, pp. 55–70 (2014)