TR: Array Abstraction with Symbolic Pivots

Nathan Wasser,
Richard Bubel
and Reiner Hähnle
TU Darmstadt
Department of Computer Science
64289 Darmstadt, Germany
Email: {wasser,bubel,haehnle}@informatik.tu-darmstadt.de

***Abstract*—In this paper we present a novel approach to automatically generate invariants for loops manipulating arrays. The intention is to achieve deductive program verification 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.**

***Index Terms*—loop invariant generation, program verification, abstract interpretation, array abstraction**

I. Introduction

Deductive program analysis and program verification must often choose a trade-off between the complexity of the properties they ascertain for a given program, 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 precision of the analysis requires addressing one of the sources for interaction (or otherwise loss of precision) with the underlying theorem prover of the analysis tool. One kind of interaction stems from the elimination of quantifiers, another is the provision of program specifications like method contracts, loop invariants or assertions that serve as hints for the theorem prover. Providing useful specifications, in particular, loop invariants is a difficult task, which requires experience and an education in writing formal specifications on the part of the user. The necessary amount of work and time hinders a wide-spread adoption pf formal verification in industry.

In this paper 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 presenting a novel approach to automatically generate invariants for loops manipulating arrays. The 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 adaptable.

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

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].

The paper is structured as follows: in Section II we introduce the logic framework and basic notions and notations needed to describe the array abstraction. Section III explains how the loop invariants are generated. In Section IV we briefly describe our implementation and demonstrate our approach along a small example. We relate our approach to the work of others in Section V and conclude the paper with Section VI giving an outlook over future research.

II. Background

A. Program Logic

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

a) Syntax: The program logic is a first order dynamic logic which is closely related to Java Card DL [2]. We begin by defining its signature which 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 sorts* $\mathcal{T}$ *together with a type hierarchy* $\preceq$*, predicates* $\mathcal{P}$*, functions* $\mathcal{F}$*, program variables* $\mathcal{PV}$ *and logical variables* $\mathcal{V}$*. The set of sorts contains at least the sorts* $\top$, `Heap`, `LocSet`, `int` *and* `int[]` *with* $\top$ *being the top element and the other sorts being direct subsorts of* $\top$.

Our logic consists of terms Trm (we 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. Note, there is a distinction between logical variables and program variables. Both are terms themselves, the difference is that logical variables must not occur in programs, but can be bound by a quantifier. On the other hand, program variables can occur in programs, but cannot be bound by a quantifier. Program variables are flexible function constants, whose value can be changed by 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 $x \in \mathcal{PV}$ and t is a term of the same type as x (or a subtype thereof). Updates can be applied to terms and formulas, i.e., given a term t then $\{\mathcal{U}\}t$ is also a term (analog 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 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 $\mathtt{p}$ terminates (uncaught exceptions are treated as non-termination) then in its final state the formula ϕ holds (our programs are deterministic). This means the box modality is used to express partial correctness. The formula $\phi \rightarrow [\mathtt{p}]\psi$ has the exact same meaning as the Hoare triple $\{\phi\}\ \mathtt{p}\ \{\psi\}$. In contrast to Hoare logic, dynamic logic allows nested modalities. The grammar for programs is:

$$\begin{aligned} \mathtt{p} \;::=\;& \mathtt{x}\ \mathtt{=}\ t \mid \mathtt{x[}t\mathtt{]}\ \mathtt{=}\ t \mid \mathtt{p;p} \\ & \mid \mathtt{if}\ (\phi)\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{p}\} \mid \mathtt{while}\ (\phi)\ \{\mathtt{p}\} \end{aligned}$$

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

We write `if (`φ`) {p}` as an abbreviation for `if (`φ`) {p} else {x = x}`, where $\mathtt{x} \in \mathcal{PV}$ is an arbitrary program variable.

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

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

1) *an* interpretation *I which assigns each*
   - *sort $T \in \mathcal{T}$ a non-empty domain $D^T \subseteq D$ s.t. for $S \preceq T \in \mathcal{T} : D^S \subseteq D^T$*
   - *$f : T_1 \times \ldots \times T_n \rightarrow T \in \mathcal{F}$ 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 $v \in \mathcal{PV}$ of type T a value $s(t) \in D^T$. We denote the set of all states by States.*

We fix the interpretation of some sorts and symbols: $I(\mathtt{int}) = \mathbb{Z}$, $I(\top) = D$ 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 to logical variable 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 sort 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 $\mathtt{p}$ to a set of states $val_{D,I,s,\beta}(\mathtt{p}) \in 2^{\mathcal{S}}$ with $val_{D,I,s,\beta}(\mathtt{p})$ being either empty or a singleton set.*

A formula φ is called valid *iff $val_{D,I,s,\beta}(\varphi) = tt$ for* 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:

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

The evaluation of a parallel update $val_{D,I,s,\beta}(\mathtt{x}_1 := t_1 \parallel \mathtt{x}_2 := t_2)$ maps a state s' to a state s'' such that s'' coincides with s' except for the program variables $\mathtt{x}_1, \mathtt{x}_2$ which are assigned the values of the terms t_i in parallel. In case of a clash between two sub-updates (i.e., when $\mathtt{x}_i = \mathtt{x}_j$ 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 $[\mathtt{p}]\varphi$ is that φ should hold in all result states of the program $\mathtt{p}$, which corresponds to partial correctness of $\mathtt{p}$ wrt. φ.

c) Heap Model.: The only heap objects we support in our program language (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 `Heap`, with two function symbols $store : \mathtt{Heap} \times \mathtt{int}[] \times \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{Heap}$ to store values on the heap and $select : \mathtt{Heap} \times \mathtt{int}[] \times \mathtt{int} \rightarrow \mathtt{int}$ to retrieve values from the heap.

For instance, $store(h, \mathtt{a}, \mathtt{i}, 3)$ returns a new heap which is identical to heap h except for the the `i`-th element of array `a` which is assigned the value 3. To retrieve the value of an array element `b[j]` we write $select(h, \mathtt{b}, \mathtt{j})$. There is a special program variable `heap` which refers to the heap accessed by programs. We abbreviate $select(\mathtt{heap}, a, i)$ to simply $a[i]$. To ease quantification about array indices, we use $\forall x \in [l..r).\phi$

[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.

as abbreviation for $\forall x.((l \leq x \wedge x < r) \rightarrow \phi))$. Further, we write $\forall x \in arr.\phi$ for $\forall x \in [0..arr.\texttt{length}).\phi$.

Closely related to heaps are location sets which are defined as terms of sort `LocSet`. Semantically, an element of `LocSet` describes a set of program locations. A program location is a pair (a, i) with $val_{D,I,s,\beta}(a) \in D^{\texttt{int}[]}, val_{D,I,s,\beta}(i) \in \mathbb{Z}$ 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 $a[0..a.\texttt{length}]$.

d) 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*) finite sets of formulas. A sequent $val_{D,I,s,\beta}(\Gamma \Rightarrow \Delta)$ has the same meaning as the formula $val_{D,I,s,\beta}(\bigwedge \Gamma \;\text{->}\; \bigvee \Delta)$. A sequent calculus *rule* is given by the rule schemata (to the right) 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 validity of the conclusion follows from the validity of all its premisses.

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

A sequent proof is a tree of which 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, here is the assignment rule for a program variable:

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

And the assignment rule for an array location:

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

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 computing its weakest precondition. Symbolic execution of *conditional statements* split the proof into two branches:

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

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

$$\textsf{loopUnwind}\quad \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}g \Rightarrow \{\mathcal{U}\}[\mathtt{p;\ while\ (}g\mathtt{)\ \{p\};\ \ r}]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\,!\,g \Rightarrow \{\mathcal{U}\}[\mathtt{r}]\varphi, \Delta\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ \ r}]\varphi, \Delta}$$

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

The first premise (initial case) ensures that the loop invariant *Inv* is valid before entering the loop. The second premise (preserves case) ensures that *Inv* is preserved by an arbitrary loop iteration, while for the third premise (use case), we have to show that after executing the remaining program, the desired postcondition φ holds. In contrast to standard loop invariants, we keep the context (Γ, Δ) in the second and third premise, following [2]. This is sound, because we use an *anonymizing update* $\mathcal{V}_{mod} = (\mathcal{V}_{mod}^{vars} \parallel \mathcal{V}_{mod}^{heap})$ 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 `LocSet` describing all array locations of $\mathtt{a}_i$ which might be changed by the loop. The anonymizing updates are:

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

where the c_i are fresh constants of the same type as $\mathtt{x}_i$ and $anonHeap_i$ are fresh constants of type `Heap`. The function $anon(h1, locset, h2)$ 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 $\mathtt{p}$ 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].

B. 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].*

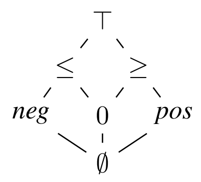
[2]The limitation to only finite ascending chains ensures termination of our approach without the need to introduce widening operators. An extension to infinite chains with widening would be easily realizable, although we have not yet deemed it necessary.

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

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



$$\begin{aligned}
\gamma(\top) &= \mathbb{Z}\\
\gamma(\leq) &= \{i \in \mathbb{Z} \mid i \leq 0\}\\
\gamma(\geq) &= \{i \in \mathbb{Z} \mid i \geq 0\}\\
\gamma(\mathit{neg}) &= \{i \in \mathbb{Z} \mid i < 0\}\\
\gamma(\mathit{pos}) &= \{i \in \mathbb{Z} \mid i > 0\}\\
\gamma(0) &= \{0\}\\
\gamma(\emptyset) &= \{\}
\end{aligned}$$

The `weakenUpdate` rule from [1] allows abstraction in our calculus:

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

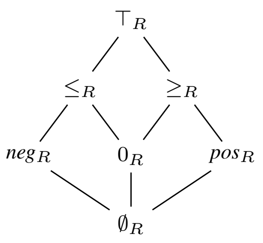
where $\bar{\mathrm{x}}$ are all program variables occurring as left-hand sides in $\mathcal{U}$ and $\bar{\mathrm{c}}$ are fresh skolem constants . The formula $\exists\bar{\gamma}.\psi$ is a shortcut for $\exists\bar{y}.(\chi_{\bar{a}}(\bar{y}) \wedge \psi[\bar{\gamma}/\bar{y}])$, where $\bar{y} = (y_1, \ldots, y_m)$ 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*.

III. Loop Invariant Generation for Arrays

We refine the value-based abstraction approach from the previous section for dealing with arrays. Rather than introducing an explicit 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 $\gamma_{\alpha_{R,i},j} \in \{arr \in \mathcal{D}^{\texttt{int[]}} \mid \forall k \in R.\chi_{\alpha_i}(arr[k])\}$.

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

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



Fixing $R = \{0, 2\}$, we have $\gamma(\geq_{\{0,2\}}) = \{arr \in \mathcal{D}^{\texttt{int[]}} \mid arr[0] \geq 0 \wedge arr[2] \geq 0\}$. 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 $[0..arr.\texttt{length})$. This range can always be taken as a fallback option if no more precise range can be found.

A. 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*:

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

where $\mathcal{U}'_{mod} := (\mathcal{U}' \parallel \mathcal{V}^{heap}_{mod})$ ($\mathcal{V}^{vars}_{mod}$ is not needed, as this is included in the abstract update $\mathcal{U}'$), $\bar{\mathrm{x}}, \bar{c}, \bar{\gamma}$ and $\exists\bar{\gamma}\varphi$ are defined as in the `weakenUpdate` rule and `old` is a fresh constant used in *Inv* to refer to the heap before loop execution.
$\mathcal{U}'$ 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. *Inv* contains invariants related to the heap. Intuitively $\mathcal{U}'_{mod}$ and *Inv* together express all states in which

$$
\begin{aligned}
\mathcal{U}' &= (\mathcal{U} \parallel \mathtt{i} := \gamma_{\geq,1} \parallel \mathtt{j} := \gamma_{\geq,2})\\
\mathit{Inv} &= (\forall k \in [0..\mathtt{j}).\ \chi_{>}(\mathtt{a}[k]))\\
&\quad \wedge\ (\forall k \in [0..\mathtt{i}).\ \chi_{\geq}(\mathtt{b}[k]))\\
&\quad \wedge\ (\forall m \in c.\ (m < 2 * i \wedge m\%2 \doteq 0)\\
&\qquad \rightarrow \chi_0(\mathtt{c}[2 * m]))\\
&\quad \wedge\ (\forall m \in c.\ \neg(m < 2 * i \wedge m\%2 \doteq 0)\\
&\qquad \rightarrow (\mathtt{c}[m] \doteq \mathit{select}(\mathtt{old}, \mathtt{c}, m)))\\
\mathcal{V}_{mod}^{heap} &= \mathtt{heap} := \mathit{anon}(\mathit{anon}(\mathtt{heap}, \mathtt{b}[0..i], \mathit{anonHeap}_1),\\
&\qquad\qquad \mathtt{c}[*], \mathit{anonHeap}_2)
\end{aligned}
$$

Fig. 1. Values for `invariantUpdate`

the program could be before or after any iteration of the loop. The first two branches ensure that the abstract update $\mathcal{U}'_{mod}$ and the invariant Inv are a valid weakening of the original update $\mathcal{U}$. The next two branches ensure that $\mathcal{U}'_{mod}$ and Inv express an 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 branch is the use case, where φ must be proven based on the state after exiting the loop and executing the remaining program. Given the program p in Listing 1, we can apply the assignment rule 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 in b[3] (cf. $\mathcal{V}_{mod}^{heap}$) are all non-negative, and *c*) the anonymized values in c are equal to their original values (if the loop does not or has not yet modified them) or are equal to 0.

Listing 1. Example

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

B. Computation of the Abstract Update and Invariants

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

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

Algorithm 1: Generating an abstract update and invariant fixpoint

```
input : the sequent $seq$
output: the fixpoint $\mathcal{U}'$ with valid $\mathcal{V}_{mod}^{heap}$ and $Inv$, as
        $(\mathcal{U}'_m, Inv)$
1  $\mathcal{U}'_m \leftarrow \mathcal{U}$;
2  while no fixpoint found do
3      /* seq is of the form:
          $\Gamma \Rightarrow \{\mathcal{U}'_m\}[\mathtt{while}\ (g)\ \{p\};\ r]\varphi, \Delta$  */
4      $\mathcal{U}^* \leftarrow \mathcal{U}'_m$; $Inv \leftarrow \Gamma \cup !\Delta$;
5      $seq \leftarrow$
          $(\Gamma, \{\mathcal{U}'_m\}g \Rightarrow \{\mathcal{U}'_m\}[p;\mathtt{while}(g)\{p\};r]\varphi, \Delta)$;
6      perform symbolic execution on $seq$;
7      /* all branches are either closed
          or loop entry reached again  */
8      foreach $\Gamma_i \Rightarrow \{\mathcal{U}_i\}[\mathtt{while}\ (g)\ \{p\};\ r]\varphi, \Delta_i$
       representing an open branch do
9          // see Def. 6 for $\dot\sqcup$
10         $(Inv, \mathcal{U}^*) \leftarrow (Inv, \mathcal{U}^*)\ \dot\sqcup\ (\Gamma_i \cup !\Delta_i, \mathcal{U}_i)$;
11     end
12     if $\mathcal{U}'_m$ is (P,Inv)-weaker than $\mathcal{U}^*$ then
13         return $(\mathcal{U}'_m, Inv)$;
14     end
15     $\mathcal{U}'_m \leftarrow \mathcal{U}^*$; $\Gamma \leftarrow \Gamma \cup \{\mathcal{U}'_m\}Inv$;
16     $seq \leftarrow (\Gamma \Rightarrow \{\mathcal{U}'_m\}[\mathtt{while}\ (g)\ \{p\};\ r]\varphi, \Delta)$;
17 end
```

In [1] a concrete implementation for joining updates $(C_1, \mathcal{U}_1) \sqcup_{abs} (C_2, \mathcal{U}_2)$ with

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

was computed as follows: For each update $\mathtt{x} := v$ in $\mathcal{U}_1$ or $\mathcal{U}_2$ the generated update is $\mathtt{x} := v$, if $\{\mathcal{U}_1\}x \doteq \{\mathcal{U}_2\}x$ under C_1, C_2 respectively. Otherwise it is $\mathtt{x} := \gamma_{\alpha_i,j}$ 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 $n \in \mathbb{N}$) $\mathtt{heap} := \gamma_{\top,n}$ for any non-identical heaps.

Definition 6 (Joining Updates)**.** *As we wish to join the heaps meaningfully, which leads to the generation of constraints, our update join operation has the signature*

$$\dot\sqcup : (2^{For} \times \mathit{Upd}) \times (2^{For} \times \mathit{Upd}) \rightarrow (2^{For} \times \mathit{Upd})$$

and is defined by the property: Let $\mathcal{U}_1$ and $\mathcal{U}_2$ be arbitrary updates in a proof P and let C_1, C_2 be formula sets representing constraints on the update values. Then for $(C, \mathcal{U}) =$

[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.

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

$(C_1,\mathcal{U}_1)\ \dot{\sqcup}\ (C_2,\mathcal{U}_2)$ *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 as shown in Algorithm 2.

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 $(C,\mathcal{U}_{res})$
1 // the heap update h' will be ignored
2 $(\mathcal{U}_{res} \parallel \texttt{heap} := h') \leftarrow (C_1,\mathcal{U}_1)\ \sqcup_{abs}\ (C_2,\mathcal{U}_2)$;
3 // see Def. 7 for $\hat{\sqcup}$
4 $(C,h) \leftarrow (C_1,\{\mathcal{U}_1\}\texttt{heap})\ \hat{\sqcup}\ (C_2,\{\mathcal{U}_2\}\texttt{heap})$;
5 $\mathcal{U}_{res} \leftarrow (\mathcal{U}_{res} \parallel \texttt{heap} := h)$;
6 return $(C,\mathcal{U}_{res})$
```

Definition 7 (Joining Heaps)**.** *The heap join operator has the signature*

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

and is defined by the property: Let h_1 *and* 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)* $(\mathcal{U} \parallel \texttt{heap} := h)$ *is (P,* C_i*)-weaker than* $(\mathcal{U} \parallel \texttt{heap} := h_i)$*, b)* $C_i \Rightarrow \{\mathcal{U} \parallel \texttt{heap} := h_i\} \bigwedge C$*, and c)* $\hat{\sqcup}$ *is associative and commutative up to first-order reasoning.*

We define the set of *normal form heaps* $\mathcal{H}_{NF} \subset Trm_{\texttt{Heap}}$ to be only those heap terms based on `heap` with any number of preceding stores and/or anonymizations. For a heap term $h \in \mathcal{H}_{NF}$ we define

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

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 $lcs(h_1,h_2)$ and all writes performed on this subheap can be given as $writes_{lcs}(h_1,h_2) := writes(h_1) \cup writes(h_2) \setminus (writes(h_1) \cap writes(h_2))$. Algorithm 3 shows how the join of heaps $(C_1,h_1)\ \hat{\sqcup}_{heap}\ (C_2,h_2)$ is calculated.

Example 3. *With a given precondition such as* $P = \forall n \in \texttt{b}.\ select(\texttt{heap},\texttt{b},n) \doteq -1$ *and the program in Listing 1, we demonstrate the first steps of Algorithm 1 with* $seq = P \Rightarrow \{\texttt{i} := 0 \parallel \texttt{j} := 0\}[\texttt{while...}]\varphi$*: After*

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

```
input : $((C_1,h_1),(C_2,h_2))$
output: the weaker constraint/heap pair $(C_{res},h_{res})$
1 $h_{res} \leftarrow lcs(h_1,h_2)$; $C_{res} \leftarrow \emptyset$;
2 $W \leftarrow writes_{lcs}(h_1,h_2)$;
3 foreach $anon(h,a[l..r],anonHeap)$ or
          $store(h,a,idx,v) \in W$ do
4     $h_{res} \leftarrow anon(h_{res},a[*],anonHeap')$;
5     $i_1,i_2 \leftarrow$ the indices of the smallest $\alpha_{i_j}$ such that
          $C_j \Rightarrow \forall k \in a.\ \chi_{\alpha_{i_j}}(select(h_j,a,k))$;
6     $C_{res} \leftarrow$
          $C_{res} \cup \{\forall k \in a.\ \chi_{\alpha_{i_1} \sqcup \alpha_{i_2}}(select(\texttt{heap},a,k))\}$
7 end
```

initialization Inv $= \{P\}$ *and* $\mathcal{U}^* = (\texttt{i} := 0 \parallel \texttt{j} := 0)$*. At line 8 of Algorithm 1 we have two open branches:*

$$\begin{gathered} P,\{\mathcal{U}^*\}g,\neg(select(\texttt{heap},\texttt{a},0) > 0) \Rightarrow \\ \{\texttt{i} := 1 \parallel \texttt{j} := 0 \parallel heap := store(store(\texttt{heap},\texttt{b},0,0),\texttt{c},0,0)\} \\ [\texttt{while...}]\varphi \end{gathered} \quad (1)$$

$$\begin{gathered} P,\{\mathcal{U}^*\}g, select(\texttt{heap},\texttt{a},0) > 0 \Rightarrow \\ \{\texttt{i} := 1 \parallel \texttt{j} := 1 \parallel heap := store(store(\texttt{heap},\texttt{b},0,1),\texttt{c},0,0)\} \\ [\texttt{while...}]\varphi \end{gathered} \quad (2)$$

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

$$\begin{aligned} C_{res} &= \{\forall m \in \texttt{b}.\ \chi_{\leq}(select(\texttt{heap},\texttt{b},m)), \\ &\qquad \forall n \in \texttt{c}.\ \chi_{\top}(select(\texttt{heap},\texttt{c},n))\} \\ h_{res} &= anon(anon(\texttt{heap},\texttt{b}[*],anonH_1),\texttt{c}[*],anonH_2) \end{aligned}$$

At line 10 of Algorithm 1 it holds that $\mathcal{U}^* = (\texttt{i} := \gamma_{\geq,1} \parallel \texttt{j} := 0 \parallel \texttt{heap} := h_{res})$ *and 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.*

C. 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, 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, due to the chosen heap join in Algorithm 3, this information is relatively weak as it assumes any update to an array element could cause a change at any index. With the generated $\mathcal{U}'$, however, we can refine $\mathcal{V}_{mod}^{heap}$ and Inv, keeping the anonymizations in $\mathcal{V}_{mod}^{heap}$ to a minimum, while producing stronger invariants Inv.

Consider the sequent $\Gamma \Rightarrow \{\mathcal{U}\}[\texttt{while}\,(g)\,\{p\}\texttt{;}\ \ r]\varphi, \Delta$. The update $(\mathcal{U}' \parallel \texttt{heap} := \{\mathcal{U}\}\texttt{heap})$ remains weaker than $\mathcal{U}$, as $\mathcal{U}'$ is weaker than $\mathcal{U}$. For the sequent $\Gamma \Rightarrow \{\mathcal{U}' \parallel \texttt{heap} := \{\mathcal{U}\}\texttt{heap}\}[\texttt{while}\,(g)\,\{p\}\texttt{;}\ \ r]\varphi, \Delta$ following Algorithm 1 we reach open branches $\Gamma_i \Rightarrow \{\mathcal{U}_i\}[\texttt{while}\ \ (g)\ \ \{p\}\texttt{;}\ \ r]\varphi, \Delta_i$. Aside from the values for $\texttt{heap}$, $\mathcal{U}'$ 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 $(\mathcal{U}^*, Inv)$, as fixpoints for these are already calculated and will not change.

When joining constraint/heap pairs we distinguish between three types of writes (see Sect. III-B): *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, for which we anonymize a (partial) range in the array and give stronger invariants.

Given a store to a variable index $store(h, a, idx, v)$, the index idx is expressible as a function $index(\gamma_{i_0,j_0}, \ldots, \gamma_{i_n,j_n})$. These γ_{i_x,j_x} can be linked to program variables in the update $\mathcal{U}'$, which contains updates $\texttt{pv}_x := \gamma_{i_x,j_x}$.We can therefore represent idx as a function $sp(\ldots \texttt{pv}_x \ldots)$.

We call $idx = sp(\ldots \texttt{pv}_x \ldots)$ a *symbolic pivot*, as it expresses what elements of the array can be changed based on which program variables and allows us to partition the array similar to pivot elements in array algorithms. Symbolic pivots split the array into an already modified partition and an unmodified partition, where (parts of) the unmodified partition may yet be modified in later iterations.

Let $P(\mathcal{W})$ be defined for an arbitrary symbolic pivot sp as: $P(\mathcal{W}) := \forall k \in [\{\mathcal{U}\}sp..\{\mathcal{W}\}sp).\ \{\mathcal{W}\}\chi_{\alpha_j}(select(\texttt{heap}, arr, k))$ Then $P(\mathcal{U})$ is trivially true, as we are quantifying over an empty set. Likewise, it is easy to show that the instance $Q(\mathcal{U})$ of the following is valid:

$$\begin{aligned} Q(\mathcal{W}) :=& \forall k \notin [\{\mathcal{U}\}sp..\{\mathcal{W}\}sp). \\ & select(\{\mathcal{W}\}\texttt{heap}, \{\mathcal{W}\}arr, k) \doteq select(\{\mathcal{U}\}\texttt{heap}, \{\mathcal{W}\}arr, k) \end{aligned}$$

Therefore, anonymizing an array arr with $anon(h, arr[*], anonHeap)$ 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 $Q(\mathcal{U}') \Rightarrow Q(\mathcal{U}_i)$. The same is true for the range $[\{\mathcal{W}\}sp..\{\mathcal{U}\}sp)$, we therefore assume in the following w.l.o.g. that $\{\mathcal{W}\}sp \geq \{\mathcal{U}\}sp$ and therefore only use the range $[\{\mathcal{U}\}sp..\{\mathcal{W}\}sp)$.

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

TABLE I
TEST SUITE RESULTS.

Method	LocSets modified	Array Invariants
arrayInit	$a[0..\texttt{i}]$	$\forall j_1 \in [0..\texttt{i}).\ \texttt{a}[j_1] \doteq 0)$
arrayMax	-	$\forall j_7 \in [0..i).\ \texttt{a}[j_7] \leq \texttt{max}$[5]
arraySplit	$b[0..\texttt{j}]$, $c[0..\texttt{k}]$	$\forall j_5 \in [0..\texttt{j}).\ \texttt{b}[j_5] > 0)$ $\forall j_6 \in [0..\texttt{k}).\ \texttt{c}[j_6] \leq 0)$
firstNotNull	-	$\forall j_0 \in [0..i).\ \texttt{a}[j_0] \doteq 0$
sentinel	-	$\forall j_{11} \in [0..i).\ \texttt{a}[j_{11}] \neq \texttt{x}$

After unwinding the loop body once we can 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. We then add the constraint $n \geq 0 \wedge (\{\mathcal{U}\}sp) + n * step \doteq v$ for a fresh n in further fixpoint iterations while ensuring $(\{\mathcal{U}'\}sp) \doteq v + step$. If this ceases to hold, sp is not iteration affine and we remove the constraint in further fixpoint iterations. Otherwise, once a fixpoint is found we know the exact array elements that may be modified, as sp is iteration affine. As expressing the affine range as a location set is non-trivial, we anonymize the entire array and create the following invariants for the modified and unmodified partitions (using the symbols of Def. 8):

$$\forall k \in arr.\ M \rightarrow P(k) \quad (3)$$

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

where $M := (k \geq \{\mathcal{U}\}sp \wedge k < sp \wedge (k - \{\mathcal{U}\}sp)\%step \doteq 0)$. Finally, we can also add invariants for array accesses which influence control flow. For each open branch with a condition $C(select(h, arr, idx))$ not already present in the sequent leading to it, we determine the symbolic pivot for idx and create an iteration affine or contiguous invariant for it. In Fig. 1 the invariants (3) and (4) are generated for the array $\texttt{c}$, while the control flow influencing access of $\texttt{a[j]}$ allows generation of an invariant for the array $\texttt{a}$.

IV. IMPLEMENTATION

The presented approach has been implemented as a proof-of-concept (available at www.key-project.org/fmcad15-albia/) and integrated into a variant of the KeY verification system for Java, which focuses on checking programs for secure information flow. In this context the requirements on the invariants is less than for functional verification and the precision of the generated invariants should be strong enough for many programs.

In addition to the array invariants we have shown can be generated for Listing 1, we created a small test suite out of some examples given in related work [6], [7] and display the resulting array invariants produced by these tests in Table I. 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

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

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. As soon as our implementation is stable, we will perform a systematic benchmarking of our approach. 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:

$$\begin{array}{l} \texttt{i} \leq \texttt{a.length} \\ \texttt{j} = \sum_{q=0}^{\texttt{i}-1} (\texttt{a}[q] > 0 \; ? \; 1 : 0) \\ \texttt{k} = \sum_{q=0}^{\texttt{i}-1} (\texttt{a}[q] > 0 \; ? \; 0 : 1) \end{array} \quad \text{, where} \quad (b \; ? \; t : t') = \begin{cases} t & \text{, if } b \\ t' & \text{, if } \neg b \end{cases}$$

V. Related Work

To find a fixpoint for non-heap variables we perform something akin to *array smashing* [8] for any array modifications in loops. Our refinements based on symbolic pivots later remedy much of this lost precision. In [9] invariants based on linear loop-dependent scalars (i.e. variables which can be modified by a loop) are computed. In [10] 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 [10]. 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. In [11] 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 [12] 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 [13] arrays are modeled as (many) contiguous partitions, while we allow both contiguous partitions as well as affine ranges. In [6] templates are used to introduce quantified formulas from quantifier-free elements, while we allow the underlying abstract domain to function as a "template." In [7] 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.

VI. Conclusion and Future Work

We presented a novel approach to generate loop invariants for loops that perform operations on arrays. The presented approach integrates nicely 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 [14] to avoid their need for user specified loop invariants.

References

[1] R. Bubel, R. Hähnle, and B. Weiß, "Abstract interpretation of symbolic execution with explicit state updates," in *7th Intl. Symposium on Formal Methods for Components and Objects (FMCO 2008)*, ser. LNCS, vol. 5751. Springer, 2009, pp. 247–277.

[2] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS. Springer, 2007, no. 4334.

[3] B. Weiß, "Deductive verification of object-oriented software — Dynamic frames, dynamic logic and predicate abstraction," Ph.D. dissertation, KIT, January 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1600837

[4] P. Rümmer, "Sequential, parallel, and quantified updates of first-order structures," in *Logic for Programming, Artificial Intelligence and Reasoning*, ser. LNCS, vol. 4246. Springer, 2006, pp. 422–436.

[5] P. Cousot and R. Cousot, "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)*. ACM, 1977, pp. 238–252.

[6] S. Gulwani, B. McCloskey, and A. Tiwari, "Lifting abstract interpreters to quantified logical domains," *SIGPLAN Not.*, vol. 43, no. 1, pp. 235–246, jan 2008. [Online]. Available: http://doi.acm.org/10.1145/1328897.1328468

[7] N. Halbwachs and M. Péron, "Discovering properties about arrays in simple programs," *SIGPLAN Not.*, vol. 43, no. 6, pp. 339–348, jun 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375623

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The essence of computation," T. A. Mogensen, D. A. Schmidt, and I. H. Sudborough, Eds. Springer, 2002, ch. Design and Implementation of a Special-purpose Static Program Analyzer for Safety-critical Real-time Embedded Software, pp. 85–108. [Online]. Available: http://dl.acm.org/citation.cfm?id=860256.860262

[9] I. Dillig, T. Dillig, and A. Aiken, "Fluid updates: Beyond strong vs. weak updates," in *Proc. of the 19th European Conf. on Programming Languages and Systems*, ser. ESOP'10. Springer, 2010, pp. 246–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11957-6_14

[10] L. Kovács and A. Voronkov, "Finding loop invariants for programs over arrays using a theorem prover," in *Proc. of the 12th Intl. Conf. on Fundamental Approaches to Software Engineering*, ser. FASE '09. Springer, 2009, pp. 470–485. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00593-0_33

[11] P. Cousot, R. Cousot, and F. Logozzo, "A parametric segmentation functor for fully automatic and scalable array content analysis," in *Proc. of the 38th Symposium on Principles of Programming Languages*, ser. POPL '11. ACM, 2011, pp. 105–118. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926399

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer, 2000, pp. 154–169.

[13] D. Gopan, T. Reps, and M. Sagiv, "A framework for numeric analysis of array operations," *SIGPLAN Not.*, vol. 40, no. 1, pp. 338–350, jan 2005. [Online]. Available: http://doi.acm.org/10.1145/1047659.1040333

[14] M. Hentschel, S. Käsdorf, R. Hähnle, and R. Bubel, "An interactive verification tool meets an IDE," in *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, 2014, pp. 55–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10181-1_4

Appendix

A. Proofs

Lemma 1. *($\sqcup_{abs}$ is an update join operator returning an empty constraint set).*

Proof. We must prove for all $C_1, C_2, C_3, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3$ that for $\mathcal{U} = (C_1,\mathcal{U}_1) \sqcup_{abs} (C_2,\mathcal{U}_2)$:

1) For $j \in \{1,2\}$ $\mathcal{U}$ is (P, C_j)-weaker than $\mathcal{U}_j$
This has already been proven in [1].
2) For $j \in \{1,2\}$ $C_j \Rightarrow \{\mathcal{U}_j\} \bigwedge \emptyset$
This is trivially true.
3) $\sqcup_{abs}$ is commutative with regard to semantics
Let $(C_2,\mathcal{U}_2) \sqcup_{abs} (C_1,\mathcal{U}_1) = \mathcal{U}_B$. Then for all program variables $\texttt{x}$ either $\{\mathcal{U}_1\}\texttt{x} = \{\mathcal{U}_2\}\texttt{x}$ in which case
$$\{\mathcal{U}\}\texttt{x} = \{\mathcal{U}_1\}\texttt{x} = \{\mathcal{U}_B\}\texttt{x}$$
or $\{\mathcal{U}\}\texttt{x}$ and $\{\mathcal{U}_B\}\texttt{x}$ are γ-symbols for an abstract element $\alpha_{i_1} \sqcup \alpha_{i_2}$ resp. $\alpha_{i_2} \sqcup \alpha_{i_1}$, which represent the same abstract element, as $\sqcup$ is a join operator on a lattice and therefore commutative.
4) $\sqcup_{abs}$ is associative with regard to semantics
Let $((C_1,\mathcal{U}_1) \sqcup_{abs} (C_2,\mathcal{U}_2)) \sqcup_{abs} (C_3,\mathcal{U}_3) = \mathcal{U}_A$ and $(C_1,\mathcal{U}_1) \sqcup_{abs} ((C_2,\mathcal{U}_2) \sqcup_{abs} (C_3,\mathcal{U}_3)) = \mathcal{U}_B$. Then for all program variables $\texttt{x}$ one of the following holds:
   a) $\{\mathcal{U}_1\}\texttt{x} = \{\mathcal{U}_2\}\texttt{x} = \{\mathcal{U}_3\}\texttt{x}$ in which case
   $$\{\mathcal{U}\}\texttt{x} = \{\mathcal{U}_1\}\texttt{x} = \{\mathcal{U}_B\}\texttt{x}$$
   b) $\{\mathcal{U}_1\}\texttt{x} = \{\mathcal{U}_2\}\texttt{x} \neq \{\mathcal{U}_3\}\texttt{x}$. Let α_a be the abstract element for $\{\mathcal{U}_1\}\texttt{x}$ and α_b be the abstract element for $\{\mathcal{U}_3\}\texttt{x}$. Then $\{\mathcal{U}\}\texttt{x}$ is a γ-symbol for the abstract element $\alpha_a \sqcup \alpha_b$, while $\{\mathcal{U}_B\}\texttt{x}$ is a γ-symbol for the abstract element $\alpha_a \sqcup (\alpha_a \sqcup \alpha_b)$. These represent the same abstract element, as $\sqcup$ is associative and idempotent.
   c) $\{\mathcal{U}_1\}\texttt{x} \neq \{\mathcal{U}_2\}\texttt{x} = \{\mathcal{U}_3\}\texttt{x}$. This is analogous to 4b.
   d) $\{\mathcal{U}_1\}\texttt{x} \neq \{\mathcal{U}_2\}\texttt{x} \neq \{\mathcal{U}_3\}\texttt{x}$ in which case $\{\mathcal{U}\}\texttt{x}$ and $\{\mathcal{U}_B\}\texttt{x}$ are γ-symbols for an abstract element $(\alpha_{i_1} \sqcup \alpha_{i_2}) \sqcup \alpha_{i_3}$ resp. $\alpha_{i_1} \sqcup (\alpha_{i_2} \sqcup \alpha_{i_3})$, which represent the same abstract element, as $\sqcup$ is a join operator on a lattice and therefore associative.

□

Lemma 2. *(The order in which the elements of W are iterated in Algorithm 3 is irrelevant for the output with regard to semantics).*

Proof. As C_{res} is a set, it is plain that the order in which elements are added to it is irrelevant. As generated elements do not rely on previously generated elements it is therefore clear that C_{res} will always be the same no matter how W is iterated through. The resulting h_{res} is influenced by the order in which the elements of W are iterated, but we show that this is not relevant with regard to semantics, i.e. $I(h_{res})$ is not influenced by the ordering. h_{res} has the form
$$anon(\ldots anon(h', a_1[0..a_1.\texttt{length}], anonH_1)\ldots,$$
$$a_n[0..a_n.\texttt{length}], anonH_n)$$
where all $anonH_i$ are fresh and therefore can be exchanged with oneanother freely without changing the semantic meaning. If $I(a_i) = I(a_j)$ for some $i < j$, we can replace $anon(h_i, a_i[0..a_i.\texttt{length}], anonH_i)$ with h_i without changing the semantic meaning, as only the outermost anonymization has an effect visible from the outside. However, as in both cases the entire array is anonymized, we could instead replace $anon(h_j, a_j[0..a_j.\texttt{length}], anonH_j)$ with h_j with the same result. We can therefore rearrange h_{res} to:
$$anon(\ldots anon(h', a_{i_1}[0..a_{i_1}.\texttt{length}], anonH_1)\ldots,$$
$$a_{i_m}[0..a_{i_m}.\texttt{length}], anonH_m)$$
where $i_j \neq i_k \rightarrow I(a_{i_j}) \neq I(a_{i_k})$. In this case the semantic meaning is unchanged if we exchange the ordering of the anonymizations, as they all refer to different arrays and can therefore be anonymized in parallel. □

Lemma 3. *($\hat{\sqcup}_{heap}$ is a heap join operator on the reduced signature $(2^{For} \times \mathcal{H}_{NF}) \times (2^{For} \times \mathcal{H}_{NF}) \rightarrow (2^{For} \times \mathcal{H}_{NF})$).*

Proof. We must prove for all $C_1, C_2, C_3, h_1, h_2, h_3$ and an arbitrary update $\mathcal{U}$ that for $(C,h) = (C_1,h_1)\ \hat{\sqcup}_{heap}\ (C_2,h_2)$:

1) For $j \in \{1,2\}$ $(\mathcal{U} \parallel \texttt{heap} := h)$ is (P, C_j)-weaker than $(\mathcal{U} \parallel \texttt{heap} := h_j)$
For all program variables $\texttt{x} \neq \texttt{heap}$ this is trivially true, as $\{\mathcal{U} \parallel \texttt{heap} := h\}\texttt{x} = \{\mathcal{U} \parallel \texttt{heap} := h_j\}\texttt{x}$. As for $\texttt{heap}$, the untouched subheap $lcs(h_1,h_2)$ in h is equal to the matching subheap in h_j, while any changes on that subheap are merely anonymizations which can obviously only weaken the heap. Therefore h is even (P, $\emptyset$)-weaker and in particular (P, C_j)-weaker than h_j.
2) For $j \in \{1,2\}$ $C_j \Rightarrow \{\mathcal{U} \parallel \texttt{heap} := h_j\} \bigwedge C$
As all elements of C are being evaluated in the update $\mathcal{U} \parallel \texttt{heap} := h_j$, they each have the form
$$\forall k \in [0..a.\texttt{length}).\ \chi_{\alpha_{i_1} \sqcup \alpha_{i_2}}(select(h_j, a, k)).$$
Furthermore, we know from line 5 in Algorithm 3 that
$$C_j \Rightarrow \forall k \in [0..a.\texttt{length}).\ \chi_{\alpha_{i_j}}(select(h_j, a, k)).$$
As $\sqcup$ is the join operator of a lattice, $\chi_{\alpha_{i_1} \sqcup \alpha_{i_2}}(x)$ must hold whenever $\chi_{\alpha_{i_j}}(x)$ holds for any x.
3) $\hat{\sqcup}_{heap}$ is commutative with regard to semantics
Let $(C_2,h_2)\ \hat{\sqcup}_{heap}\ (C_1,h_1) = (C_B,h_B)$. With Lemma 2 we can choose the ordering such that h is identical to h_B. Further, for each element $F \in C$ ($F \in C_B$), F has the form
$$\forall k \in [0..a.\texttt{length}).\ \chi_{\alpha_{i_1} \sqcup \alpha_{i_2}}(select(\texttt{heap}, a, k))$$
and there is a matching formula $G \in C_B$ ($G \in C$):
$$\forall k \in [0..a.\texttt{length}).\ \chi_{\alpha_{i_2} \sqcup \alpha_{i_1}}(select(\texttt{heap}, a, k))$$

As $\sqcup$ is the join operator of a lattice, it is commutative and therefore the formulas are equivalent.

4) $\hat{\sqcup}_{heap}$ is associative with regard to semantics
Let $((C_1, h_1) \hat{\sqcup}_{heap} (C_2, h_2)) \hat{\sqcup}_{heap} (C_3, h_3) = (C_A, h_A)$ and $(C_1, h_1) \hat{\sqcup}_{heap} ((C_2, h_2) \hat{\sqcup}_{heap} (C_3, h_3)) = (C_B, h_B)$. With Lemma 2 we can choose the ordering such that h_A is identical to h_B. Further, for each element $F \in C_A$, F has the form

$$\forall k \in [0..a.\texttt{length}).\ \chi_{(\alpha_{i_1} \sqcup \alpha_{i_2}) \sqcup \alpha_{i_3}}(select(\texttt{heap}, a, k))$$

and there is a matching formula $G \in C_B$:

$$\forall k \in [0..a.\texttt{length}).\ \chi_{\alpha_{i_1} \sqcup (\alpha_{i_2} \sqcup \alpha_{i_3})}(select(\texttt{heap}, a, k))$$

As $\sqcup$ is the join operator of a lattice, it is associative and therefore the formulas are equivalent. This holds analogously for all $F' \in C_B$.

□

Lemma 4. *($\dot{\sqcup}_{upd}$ is an update join operator).*

Proof. We must prove for all $C_1, C_2, C_3, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3$ that for $(C, \mathcal{U}) = (C_1, \mathcal{U}_1) \dot{\sqcup}_{upd} (C_2, \mathcal{U}_2)$:

1) For $j \in \{1, 2\}$ $\mathcal{U}$ is (P, C_j)-weaker than $\mathcal{U}_j$
For all program variables $\texttt{x} \neq \texttt{heap}$ we have:

$$\{\mathcal{U}\}\texttt{x} = \{(C_1, \mathcal{U}_1) \sqcup_{abs} (C_2, \mathcal{U}_2)\}\texttt{x}$$

From Lemma 1 we know that $(C_1, \mathcal{U}_1) \sqcup_{abs} (C_2, \mathcal{U}_2)$ is (P, C_j)-weaker than $\mathcal{U}_j$. Further, we know that $(C_1, \{\mathcal{U}_1\}\texttt{heap}) \hat{\sqcup}_{heap} (C_2, \{\mathcal{U}_2\}\texttt{heap}) = (C, \{\mathcal{U}\}\texttt{heap})$. We therefore know that for some updates $\mathcal{U}^*, \mathcal{U}_j^*$ it holds that $\mathcal{U} = (\mathcal{U}^* \parallel \texttt{heap} := h)$ is (P, C_j)-weaker than $(\mathcal{U}^* \parallel \texttt{heap} := h_j)$, which is (P, C_j)-weaker than $(\mathcal{U}_j^* \parallel \texttt{heap} := h_j) = \mathcal{U}_j$. As (P, C_j)-weaker is transitive, it follows that $\mathcal{U}$ is (P, C_j)-weaker than $\mathcal{U}_j$.

2) For $j \in \{1, 2\}$ $C_j \Rightarrow \{\mathcal{U}_j\} \bigwedge C$
As $(C, h) = (C_1, \{\mathcal{U}_1\}\texttt{heap}) \hat{\sqcup} (C_2, \{\mathcal{U}_2\}\texttt{heap})$ and $\hat{\sqcup}$ is a heap join operator, we know that

$$C_j \Rightarrow \{\mathcal{U}^* \parallel \texttt{heap} := h_j\} \bigwedge C$$

for $h_j = \{\mathcal{U}_j\}\texttt{heap}$ and all updates $\mathcal{U}^*$. As the update $\mathcal{U}_j$ is equivalent to an update $(\mathcal{U}' \parallel \texttt{heap} := h_j)$ for some update $\mathcal{U}'$, it therefore must also hold that:

$$C_j \Rightarrow \{\mathcal{U}_j\} \bigwedge C.$$

3) $\dot{\sqcup}_{upd}$ is commutative with regard to semantics
Let $(C_2, \mathcal{U}_2) \dot{\sqcup}_{upd} (C_1, \mathcal{U}_1) = (C_B, \mathcal{U}_B \parallel \texttt{heap} := h_B)$ and $\mathcal{U} = (\mathcal{U}_A \parallel \texttt{heap} := h_A)$. Then from line 4 of Algorithm 2 it follows that

$$\begin{aligned} (C, h_A) &= (C_1, \{\mathcal{U}_1\}\texttt{heap}) \hat{\sqcup} (C_2, \{\mathcal{U}_2\}\texttt{heap}) \\ (C_B, h_B) &= (C_2, \{\mathcal{U}_2\}\texttt{heap}) \hat{\sqcup} (C_1, \{\mathcal{U}_1\}\texttt{heap}) \end{aligned}$$

As $\hat{\sqcup}$ is a heap join operator (C, h_A) and (C_B, h_B) are equivalent with regard to semantics. For any program variable $\texttt{x} \neq \texttt{heap}$:

$$\begin{aligned} \{\mathcal{U}_{\mathcal{A}}\}\texttt{x} &= \{(C_1, \mathcal{U}_1) \sqcup_{abs} (C_2, \mathcal{U}_2)\}\texttt{x} \\ \{\mathcal{U}_{\mathcal{B}}\}\texttt{x} &= \{(C_2, \mathcal{U}_2) \sqcup_{abs} (C_1, \mathcal{U}_1)\}\texttt{x} \end{aligned}$$

As $\sqcup_{abs}$ is an update join $\mathcal{U}_{\mathcal{A}}$ and $\mathcal{U}_{\mathcal{B}}$ are also equivalent with regard to semantics. As C and C_B are equivalent and the updates $(\mathcal{U}_A \parallel \texttt{heap} := h_A)$ and $(\mathcal{U}_B \parallel \texttt{heap} := h_B)$ behave equally, $\dot{\sqcup}_{upd}$ is commutative with regard to semantics.

4) $\dot{\sqcup}_{upd}$ is associative with regard to semantics
Let $((C_1, \mathcal{U}_1) \dot{\sqcup}_{upd} (C_2, \mathcal{U}_2)) \dot{\sqcup}_{upd} (C_3, \mathcal{U}_3) = (C_A, \mathcal{U}_A)$ and $(C_1, \mathcal{U}_1) \dot{\sqcup}_{upd} ((C_2, \mathcal{U}_2) \dot{\sqcup}_{upd} (C_3, \mathcal{U}_3)) = (C_B, \mathcal{U}_B)$ for some updates $\mathcal{U}_A^*, \mathcal{U}_B^*$ with $\mathcal{U}_A = (\mathcal{U}_A^* \parallel \texttt{heap} := h_A)$ and $\mathcal{U}_B = (\mathcal{U}_B^* \parallel \texttt{heap} := h_B)$. Then for all program variables $\texttt{x} \neq \texttt{heap}$:

$$\begin{aligned} \{\mathcal{U}_A^*\}\texttt{x} &= \{((C_1, \mathcal{U}_1) \sqcup_{abs} (C_2, \mathcal{U}_2)) \sqcup_{abs} (C_3, \mathcal{U}_3)\}\texttt{x} \\ \{\mathcal{U}_B^*\}\texttt{x} &= \{(C_1, \mathcal{U}_1) \sqcup_{abs} ((C_2, \mathcal{U}_2) \sqcup_{abs} (C_3, \mathcal{U}_3))\}\texttt{x} \end{aligned}$$

With regard to semantics $\mathcal{U}_A^*$ and $\mathcal{U}_B^*$ are equivalent as $\sqcup_{abs}$ is associative. With $h_i = \{\mathcal{U}_i\}\texttt{heap}$ for $i \in \{1, 2, 3\}$ we have:

$$\begin{aligned} (C_A, h_A) &= ((C_1, h_1) \hat{\sqcup} (C_2, h_2)) \hat{\sqcup} (C_3, h_3) \\ (C_B, h_B) &= (C_1, h_1) \hat{\sqcup} ((C_2, h_2) \hat{\sqcup} (C_3, h_3)) \end{aligned}$$

So (C_A, h_A) and (C_B, h_B) are also equivalent as $\hat{\sqcup}$ is associative with regard to semantics. Therefore $\dot{\sqcup}_{upd}$ is associative with regard to semantics.

□