Permission Inference for Array Programs

Jérôme Dohrau, Alexander J. Summers, Caterina Urban,
Severin Münger, and Peter Müller[0000−0001−7001−2566]

Department of Computer Science, ETH Zurich, Switzerland
jerome.dohrau@inf.ethz.ch severin.muenger@alumni.ethz.ch
{alexander.summers, caterina.urban, peter.mueller}@inf.ethz.ch

Abstract. Information about the memory locations accessed by a program is, for instance, required for program parallelisation and program verification. Existing inference techniques for this information provide only partial solutions for the important class of array-manipulating programs. In this paper, we present a static analysis that infers the memory footprint of an array program in terms of permission pre- and postconditions as used, for example, in separation logic. This formulation allows our analysis to handle concurrent programs and produces specifications that can be used by verification tools. Our analysis expresses the permissions required by a loop via maximum expressions over the individual loop iterations. These maximum expressions are then solved by a novel maximum elimination algorithm, in the spirit of quantifier elimination. Our approach is sound and is implemented; an evaluation on existing benchmarks for memory safety of array programs demonstrates accurate results, even for programs with complex access patterns and nested loops.

1 Introduction

Information about the memory locations accessed by a program is crucial for many applications such as static data race detection [45], code optimisation [26, 33, 16], program parallelisation [17, 5], and program verification [30, 23, 39, 38]. The problem of inferring this information statically has been addressed by a variety of static analyses, e.g., [9, 42]. However, prior works provide only partial solutions for the important class of array-manipulating programs for at least one of the following reasons. (1) They approximate the entire array as one single memory location [4] which leads to imprecise results; (2) they do not produce specifications, which are useful for several important applications such as human inspection, test case generation, and especially deductive program verification; (3) they are limited to sequential programs.

In this paper, we present a novel analysis for array programs that addresses these shortcomings. Our analysis employs the notion of *access permission* from separation logic and similar program logics [40, 43]. These logics associate a permission with each memory location and enforce that a program part accesses a location only if it holds the associated permission. In this setting, determining the accessed locations means to infer a sufficient precondition that specifies the permissions required by a program part.

Phrasing the problem as one of permission inference allows us to address the three problems mentioned above. (1) We distinguish different array elements by tracking the permission for each element separately. (2) Our analysis infers pre- and postconditions for both methods and loops and emits them in a form that can be used by verification tools. The inferred specifications can easily be complemented with permission specifications for non-array data structures and with functional specifications. (3) We support concurrency in three important ways. First, our analysis is sound for concurrent program executions because permissions guarantee that program executions are data race free and reduce thread interactions to specific points in the program such as forking or joining a thread, or acquiring or releasing a lock. Second, we develop our analysis for a programming language with primitives that represent the ownership transfer that happens at these thread interaction points. These primitives, `inhale` and `exhale` [31, 38], express that a thread obtains permissions (for instance, by acquiring a lock) or loses permissions (for instance, by passing them to another thread along with a message) and can thereby represent a wide range of thread interactions in a uniform way [32, 44]. Third, our analysis distinguishes read and write access and, thus, ensures exclusive writes while permitting concurrent read accesses. As is standard, we employ *fractional permissions* [6] for this purpose; a full permission is required to write to a location, but any positive fraction permits read access.

Approach. Our analysis reduces the problem of reasoning about permissions for array elements to reasoning about numerical values for permission fractions. To achieve this, we represent permission fractions for all array elements $q_a[q_i]$ using a *single* numerical expression $t(q_a, q_i)$ parameterised by q_a and q_i. For instance, the conditional term $(q_a{=}\mathsf{a} \wedge q_i{=}\mathsf{j}\ ?\ 1 : 0)$ represents full permission (denoted by 1) for array element `a[j]` and no permission for all other array elements.

Our analysis employs a *precise* backwards analysis for *loop-free* code: a variation on the standard notion of weakest preconditions. We apply this analysis to loop bodies to obtain a permission precondition for a single loop iteration. Per array element, the *whole loop* requires the *maximum* fraction over all loop iterations, adjusted by permissions gained and lost during loop execution. Rather than computing permissions via a fixpoint iteration (for which a precise widening operator is difficult to design), we express them as a maximum over the variables changed by the loop execution. We then use inferred numerical invariants on these variables and a novel *maximum elimination* algorithm to infer a specification for the entire loop. Permission postconditions are obtained analogously.

For the method `copyEven` in Fig. 1, the analysis determines that the permission amount required by a single loop iteration is $(\mathsf{j}\%2{=}0?(q_a{=}\mathsf{a} \wedge q_i{=}\mathsf{j}?\mathsf{rd}{:}0){:}(q_a{=}\mathsf{a} \wedge q_i{=}\mathsf{j}\ ?\ 1 : 0))$. The symbol `rd` represents a fractional read permission. Using a suitable integer invariant for the loop counter `j`, we obtain the loop precondition $\max_{\mathsf{j}|0\leq\mathsf{j}<\mathtt{len(a)}}((\mathsf{j}\%2{=}0\ ?\ (q_a{=}\mathsf{a} \wedge q_i{=}\mathsf{j}\ ?\ \mathsf{rd} : 0) : (q_a{=}\mathsf{a} \wedge q_i{=}\mathsf{j}\ ?\ 1 : 0)))$. Our maximum elimination algorithm obtains $(q_a{=}\mathsf{a} \wedge 0{\leq}q_i{<}\mathtt{len}(\mathsf{a})\ ?\ (q_i\%2{=}0\ ?\ \mathsf{rd} : 1) : 0)$. By ranging over all q_a and q_i, this can be read as read permission for even indices and write permission for odd indices within the array `a`'s bounds.

```
method copyEven(a: Int[]) {
   var j, v: Int := 0;
   while(j < length(a)) {
      if (j % 2 == 0) { v := a[j] }
      else { a[j] := v };
      j := j + 1
   }
}
```

Fig. 1: Program `copyEven`.

```
method parCopyEven(a: Int[]) {
   var j: Int := 0;
   while(j < length(a)/2) {
      exhale(a, 2*j, 1/2);
      exhale(a, 2*j+1, 1);
      j := j + 1
   }
}
```

Fig. 2: Program `parCopyEven`.

$$
\begin{aligned}
e &::= n \mid x \mid n{\cdot}x \mid e_1 + e_2 \mid e_1 - e_2 \mid a[e] \mid \texttt{len}(a) \mid (b\,?\,e_1 : e_2) \\
b &::= e_1\, op\, e_2 \mid e\%n{=}0 \mid e\%n{\neq}0 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \\
op &\in \{=, \neq, <, \leq, >, \geq\} \\
p &::= q \mid \mathsf{rd} \mid p_1 + p_2 \mid p_1 - p_2 \mid \min(p_1, p_2) \mid \max(p_1, p_2) \mid (b\,?\,p_1 : p_2) \\
s &::= \texttt{skip} \mid x{:=}e \mid a_1{:=}a_2 \mid x{:=}a[e] \mid a[e_1]{:=}e_2 \mid \texttt{exhale}(a, e, p) \mid \texttt{inhale}(a, e, p) \\
&\quad \mid (s_1; s_2) \mid \texttt{if}(b)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \} \mid \texttt{while}\ (b)\ \{\ s\ \}
\end{aligned}
$$

Fig. 3: Programming Language. n ranges over integer constants, x over integer variables, a over array variables, q over non-negative fractional (permission-typed) constants. e stands for integer expressions, and b for boolean. Permission expressions p are a separate syntactic category.

Contributions. The contributions of our paper are:

1. A novel permission inference that uses maximum expressions over parameterised arithmetic expressions to summarise loops (Sec. 3 and Sec. 4)
2. An algorithm for eliminating maximum (and minimum) expressions over an unbounded number of cases (Sec. 5)
3. An implementation of our analysis, which will be made available as an artifact
4. An evaluation on benchmark examples from existing papers and competitions, demonstrating that we obtain sound, precise, and compact specifications, even for challenging array access patterns and parallel loops (Sec. 6)
5. Proof sketches for the soundness of our permission inference and correctness of our maximum elimination algorithm (in the technical report (TR) [**?**])

2 Programming Language

We define our inference technique over the programming language in Fig. 3. Programs operate on integers (expressions e), booleans (expressions b), and one-dimensional integer arrays (variables a); a generalisation to other forms of arrays is straightforward and supported by our implementation. Arrays are read and updated via the statements $x{:=}a[e]$ and $a[e]{:=}x$; array lookups in expressions are not part of the surface syntax, but are used internally by our analysis. Permission expressions p evaluate to rational numbers; rd, min, and max are for internal use.

A full-fledged programming language contains many statements that affect the ownership of memory locations, expressed via permissions [32, 44]. For example

in a concurrent setting, a fork operation may transfer permissions to the new thread, acquiring a lock obtains permission to access certain memory locations, and messages may transfer permissions between sender and receiver. Even in a sequential setting, the concept is useful: in procedure-modular reasoning, a method call transfers permissions from the caller to the callee, and back when the callee terminates. Allocation can be represented as obtaining a fresh object and then obtaining permission to its locations.

For the purpose of our permission inference, we can reduce all of these operations to two basic statements that directly manipulate the permissions currently held [31, 38]. An `inhale`(a, e, p) statement adds the amount p of permission for the array location $a[e]$ to the currently held permissions. Dually, an `exhale`(a, e, p) statement requires that this amount of permission is *already* held, and then removes it. We assume that for any `inhale` or `exhale` statements, the permission expression p denotes a non-negative fraction. For simplicity, we restrict `inhale` and `exhale` statements to a *single* array location, but the extension to unboundedly-many locations from the same array is straightforward [37].

Semantics. The operational semantics of our language is mostly standard, but is instrumented with additional state to track how much permission is held to each heap location; a program state therefore consists of a triple of heap H (mapping pairs of array identifier and integer index to integer values), a *permission map* P, mapping such pairs to *permission amounts*, and an environment σ mapping variables to values (integers or array identifiers).

The execution of `inhale` or `exhale` statements causes modifications to the permission map, and all array accesses are guarded with checks that *at least some* permission is held when reading and that full (1) permission is held when writing [6]. If these checks (or an `exhale` statement) fail, the execution terminates with a *permission failure*. Permission amounts greater than 1 indicate invalid states that cannot be reached by a program execution. We model run-time errors other than permission failures (in particular, out-of-bounds accesses) as stuck configurations.

3 Permission Inference for Loop-Free Code

Our analysis infers a sufficient permission precondition and a guaranteed permission postcondition for each method of a program. Both conditions are mappings from array elements to permission amounts. Executing a statement s in a state whose permission map P contains at least the permissions required by a *sufficient permission precondition* for s is guaranteed to not result in a permission failure. A *guaranteed permission postcondition* expresses the permissions that will at least be held when s terminates (see Sec. A of the TR [**?**] for formal definitions).

In this section, we define inference rules to compute sufficient permission preconditions for loop-free code. For programs which do not add or remove permissions via `inhale` and `exhale` statements, the same permissions will still

$$
\begin{array}{c}
\mathit{pre}(\texttt{skip}, p) = p \qquad \mathit{pre}((s_1; s_2), p) = \mathit{pre}(s_1, \mathit{pre}(s_2, p)) \\
\mathit{pre}(x{:=}e, p) = p[e/x] \quad \mathit{pre}(x{:=}a[e], p) = \max(p[a[e]/x], \alpha_{a,e}(\mathsf{rd})) \\
\mathit{pre}(a[e]{:=}x, p) = \max(p[a'[e'] \mapsto (e = e' \wedge a = a' \,?\, x : a'[e'])], \alpha_{a,e}(1)) \\
\mathit{pre}(\texttt{exhale}(a, e, p'), p) = p + \alpha_{a,e}(p') \quad \mathit{pre}(\texttt{inhale}(a, e, p'), p) = \max(0, p - \alpha_{a,e}(p')) \\
\mathit{pre}(\texttt{if}(b)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, p) = (b \,?\, \mathit{pre}(s_1, p) : \mathit{pre}(s_2, p))
\end{array}
$$

$$
\begin{array}{c}
\Delta(\texttt{skip}, p) = p \qquad \Delta((s_1; s_2), p) = \Delta(s_1, \Delta(s_2, p)) \\
\Delta(x{:=}e, p) = p[e/x] \quad \Delta(x{:=}a[e], p) = p[a[e]/x] \\
\Delta(a[e]{:=}x, p) = p[a'[e'] \mapsto (e = e' \wedge a = a' \,?\, x : a'[e'])] \\
\Delta(\texttt{exhale}(a, e, p'), p) = p - \alpha_{a,e}(p') \quad \Delta(\texttt{inhale}(a, e, p'), p) = p + \alpha_{a,e}(p') \\
\Delta(\texttt{if}(b)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, p) = (b \,?\, \Delta(s_1, p) : \Delta(s_2, p))
\end{array}
$$

Fig. 4: The backwards analysis rules for permission preconditions and relative permission differences. The notation $\alpha_{a,e}(p)$ is a shorthand for $(q_a{=}a \wedge q_i{=}e \,?\, p{:}0)$ and denotes p permission for the array location $a[e]$. Moreover, $p[a'[e'] \mapsto e]$ matches all array accesses in p and replaces them with the expression obtained from e by substituting all occurrences of a' and e' with the matched array and index, respectively. The cases for inhale statements are slightly simplified; the full rules are given in Fig. 6 of the TR [**?**].

be held after executing the code; however, to infer guaranteed permission post-conditions in the general case, we also infer the difference in permissions between the state before and after the execution. We will discuss loops in the next section. Non-recursive method calls can be handled by applying our analysis bottom-up in the call graph and using `inhale` and `exhale` statements to model the permission effect of calls. Recursion can be handled similarly to loops, but is omitted here.

We define our permission analysis to track and generate *permission expressions* parameterised by two distinguished variables q_a and q_i; by parameterising our expressions in this way, we can use a single expression to represent a permission amount for each pair of q_a and q_i values.

Preconditions. The *permission precondition* of a loop-free statement s and a postcondition permission p (in which q_a and q_i potentially occur) is denoted by $\mathit{pre}(s, p)$, and is defined in Fig. 4. Most rules are straightforward adaptations of a classical weakest-precondition computation. Array lookups require some permission to the accessed array location; we use the internal expression rd to denote a non-zero permission amount; a post-processing step can later replace rd by a concrete rational. Since downstream code may require further permission for this location, represented by the permission expression p, we take the maximum of both amounts. Array updates require full permission and need to take aliasing into account. The case for `inhale` subtracts the inhaled permission amount from the permissions required by downstream code; the case for `exhale` adds the permissions to be exhaled. Note that this addition may lead to a required permission

amount exceeding the full permission. This indicates that the statement is not feasible, that is, all executions will lead to a permission failure.

To illustrate our *pre* definition, let s be the body of the loop in the `parCopyEven` method in Fig. 2. The precondition $pre(s, 0) = (q_a{=}\texttt{a} \wedge q_i = 2{*}\texttt{j}\,?\,1/2{:}0){+}(q_a{=}\texttt{a} \wedge q_i{=}2{*}\texttt{j}{+}1\,?\,1:0)$ expresses that a loop iteration requires a half permission for the even elements of array `a` and full permission for the odd elements.

Postconditions. The final state of a method execution includes the permissions held in the method pre-state, adjusted by the permissions that are inhaled or exhaled during the method execution. To perform this adjustment, we compute the difference in permissions before and after executing a statement. The *relative permission difference* for a loop-free statement s and a permission expression p (in which q_a and q_i potentially occur) is denoted by $\Delta(s, p)$, and is defined backward, analogously to *pre* in Fig. 4. The second parameter p acts as an accumulator; the difference in permission is represented by evaluating $\Delta(s, 0)$.

For a statement s with precondition $pre(s, 0)$, we obtain the postcondition $pre(s, 0){+}\Delta(s, 0)$. Let s again be the loop body from `parCopyEven`. Since s contains **`exhale`** statements, we obtain $\Delta(s, 0) = 0 - (q_a{=}\texttt{a} \wedge q_i{=}2{*}\texttt{j}\,?\,1/2:0) - (q_a{=}\texttt{a} \wedge q_i{=}2{*}\texttt{j}{+}1\,?\,1:0)$. Thus, the postcondition $pre(s, 0) + \Delta(s, 0)$ can be simplified to 0. This reflects the fact that all required permissions for a single loop iteration are lost by the end of its execution.

Since our Δ operator performs a backward analysis, our permission postconditions are expressed in terms of the pre-state of the execution of s. To obtain classical postconditions, any heap accesses need to refer to the pre-state heap, which can be achieved in program logics by using **`old`** expressions or logical variables. Formalizing the postcondition inference as a backward analysis simplifies our treatment of loops and has technical advantages over classical strongest-postconditions, which introduce existential quantifiers for assignment statements. A limitation of our approach is that our postconditions cannot capture situations in which a statement obtains permissions to locations for which no pre-state expression exists, e.g. allocation of new arrays. Our postconditions are sound; to make them precise for such cases, our inference needs to be combined with an additional forward analysis, which we leave as future work.

4 Handling Loops via Maximum Expressions

In this section, we first focus on obtaining a sufficient permission precondition for the execution of a loop in isolation (independently of the code after it) and then combine the inference for loops with the one for loop-free code described above.

4.1 Sufficient Permission Preconditions for Loops

A sufficient permission precondition for a loop guarantees the absence of permission failures for a potentially unbounded number of executions of the loop body. This concept is different from a loop invariant: we require a precondition for

all executions of a particular loop, but it need not be inductive. Our technique obtains such a loop precondition by projecting a permission precondition for a single loop iteration over all possible initial states for the loop executions.

Exhale-Free Loop Bodies. We consider first the simpler (but common) case of a loop that does not contain `exhale` statements, e.g., does not transfer permissions to a forked thread. The solution for this case is also sound for loop bodies where each `exhale` is followed by an `inhale` for the same array location and at least the same permission amount, as in the encoding of most method calls.

Consider a sufficient permission precondition p for the body of a loop `while` (b) { s }. By definition, p will denote sufficient permissions to execute s once; the precise locations to which p requires permission depend on the initial state of the loop iteration. For example, the sufficient permission precondition for the body of the `copyEven` method in Fig. 1, $(\mathtt{j}\%2{=}0\,?\,(q_a{=}\mathtt{a} \wedge q_i{=}\mathtt{j}\,?\,\mathtt{rd}:0):(q_a{=}\mathtt{a} \wedge q_i{=}\mathtt{j}\,?\,1:0))$, requires permissions to different array locations, depending on the value of `j`. To obtain a sufficient permission precondition for the entire loop, we leverage an *over-approximating* loop invariant $\mathcal{I}^+$ from an off-the-shelf numerical analysis (e.g., [14]) to over-approximate all possible values of the numerical variables that get assigned in the loop body, here, `j`. We can then express the loop precondition using the *pointwise maximum* $\max_{\mathtt{j}|\mathcal{I}^+ \wedge b}(p)$, over the values of `j` that satisfy the condition $\mathcal{I}^+ \wedge b$. (The maximum over an empty range is defined to be 0.) For the `copyEven` method, given the invariant $0 \leq \mathtt{j} \leq \mathtt{len(a)}$, the loop precondition is $\max_{\mathtt{j}|0\leq\mathtt{j}<\mathtt{len(a)}}(p)$.

In general, a permission precondition for a loop body may also depend on array *values*, e.g., if those values are used in branch conditions. To avoid the need for an expensive array value analysis, we define both an over- and an under-approximation of permission expressions, denoted $p^{\uparrow}$ and $p^{\downarrow}$ (cf. Sec. A.1 of the TR [**?**]), with the guarantees that $p \leq p^{\uparrow}$ and $p^{\downarrow} \leq p$. These approximations abstract away array-dependent conditions, and have an impact on precision only when array values are used to determine a location to be accessed. For example, a linear array search for a particular value accesses the array only up to the (a-priori unknown) point at which the value is found, but our permission precondition conservatively requires access to the full array.

Theorem 1. *Let* `while` (b) { s } *be an exhale-free loop, let* $\overline{x}$ *be the integer variables modified by* s*, and let* $\mathcal{I}^+$ *be a sound over-approximating numerical loop invariant (over the integer variables in* s*). Then* $\max_{\overline{x}|\mathcal{I}^+ \wedge b}(pre(s,0)^{\uparrow})$ *is a sufficient permission precondition for* `while` (b) { s }.

Loops with Exhale Statements. For loops that contain `exhale` statements, the approach described above does not always guarantee a sufficient permission precondition. For example, if a loop gives away full permission to the *same* array location in every iteration, our pointwise maximum construction yields a precondition requiring the full permission once, as opposed to the *unsatisfiable* precondition (since the loop is guaranteed to cause a permission failure).

As explained above, our inference is sound if each `exhale` statement is followed by a corresponding `inhale`, which can often be checked syntactically. In the following, we present another decidable condition that guarantees soundness and that can be checked efficiently by an SMT solver. If neither condition holds, we preserve soundness by inferring an unsatisfiable precondition; we did not encounter any such examples in our evaluation.

Our soundness condition checks that the maximum of the permissions required by two loop iterations is not less than the permissions required by executing the two iterations in sequence. Intuitively, that is the case when neither iteration removes permissions that are required by the other iteration.

Theorem 2 (Soundness Condition for Loop Preconditions). *Given a loop* `while` (b) $\{\ s\ \}$*, let* $\overline{x}$ *be the integer variables modified in* s *and let* $\overline{v}$ *and* $\overline{v'}$ *be two fresh sets of variables, one for each of* $\overline{x}$*. Then* $\max_{\overline{x}|\mathcal{I}^+\wedge b}(pre(s,0)^\uparrow)$ *is a sufficient permission precondition for* `while` (b) $\{\ s\ \}$ *if the following implication is valid in all states:*

$$(\mathcal{I}^+ \wedge b)\overline{[v/x]} \ \wedge\ (\mathcal{I}^+ \wedge b)\overline{[v'/x]} \ \wedge\ (\bigvee \overline{v \neq v'}) \ \Rightarrow$$
$$\max(pre(s,0)^\uparrow\overline{[v/x]}, pre(s,0)^\uparrow\overline{[v'/x]}) \geq pre(s, pre(s,0)^\uparrow\overline{[v'/x]})^\uparrow\overline{[v/x]}$$

The additional variables $\overline{v}$ and $\overline{v'}$ are used to model two arbitrary valuations of $\overline{x}$; we constrain these to represent two initial states allowed by $\mathcal{I}^+ \wedge b$ and different from each other for at least one program variable. We then require that the effect of analysing each loop iteration independently and taking the maximum is not smaller than the effect of sequentially composing the two loop iterations.

The theorem requires implicitly that no two different iterations of a loop observe exactly the same values for all integer variables. If that could be the case, the condition $\bigvee \overline{v \neq v'}$ would cause us to ignore a potential pair of initial states for two different loop iterations. To avoid this problem, we assume that all loops satisfy this requirement; it can easily be enforced by adding an additional variable as loop iteration counter [21].

For the `parCopyEven` method (Fig. 2), the soundness condition holds since, due to the $v \neq v'$ condition, the two terms on the right of the implication are equal for all values of q_i. We can thus infer a sufficient precondition as $\max_{\texttt{j}|0\leq\texttt{j}<\texttt{len(a)}/2}((q_a{=}\texttt{a} \wedge q_i = 2{*}\texttt{j}\ ?\ 1/2 : 0) + (q_a{=}\texttt{a} \wedge q_i{=}2{*}\texttt{j}{+}1\ ?\ 1 : 0))$.

4.2 Permission Inference for Loops

We can now extend the pre- and postcondition inference from Sec. 3 with loops. $pre($`while` (b) $\{\ s\ \}, p)$ must require permissions such that (1) the loop executes without permission failure and (2) at least the permissions described by p are held when the loop terminates. While the former is provided by the loop precondition as defined in the previous subsection, the latter also depends on the permissions gained or lost during the execution of the loop. To characterise these permissions, we extend the Δ operator from Sec. 3 to handle loops.

Under the soundness condition from Thm. 2, we can mimic the approach from the previous subsection and use over-approximating invariants to project

out the permissions *lost* in a single loop iteration (where $\Delta(s,0)$ is negative) to those lost by the entire loop, using a maximum expression. This projection conservatively assumes that the permissions lost in a single iteration are lost by all iterations whose initial state is allowed by the loop invariant and loop condition. This approach is a sound over-approximation of the permissions *lost*.

However, for the permissions *gained* by a loop iteration (where $\Delta(s,0)$ is positive), this approach would be unsound because the over-approximation includes iterations that may not actually happen and, thus, permissions that are not actually gained. For this reason, our technique handles gained permissions via an *under-approximate*[1] numerical loop invariant $\mathcal{I}^-$ (e.g., [35]) and thus projects the gained permissions only over iterations that will surely happen.

This approach is reflected in the definition of our Δ operator below via d, which represents the permissions *possibly lost* or *definitely gained* over all iterations of the loop. In the former case, we have $\Delta(s,0) < 0$ and, thus, the first summand is 0 and the computation based on the over-approximate invariant applies (note that the negated maximum of negated values is the minimum; we take the minimum over negative values). In the latter case ($\Delta(s,0) > 0$), the second summand is 0 and the computation based on the under-approximate invariant applies (we take the maximum over positive values).

$$\begin{aligned}\Delta(\texttt{while}\ (b)\ \{\ s\ \},p) &= (b\ ?\ d + p' : p),\ \textit{where:}\\ d &= \max_{\overline{x}|\mathcal{I}^- \wedge b} \max(0, \Delta(s,0))^{\downarrow} - \max_{\overline{x}|\mathcal{I}^+ \wedge b} \max(0, -\Delta(s,0))^{\uparrow}\\ p' &= \max_{\overline{x}|\mathcal{I}^- \wedge \neg b} \max(0,p)^{\downarrow} - \max_{\overline{x}|\mathcal{I}^+ \wedge \neg b} \max(0,-p)^{\uparrow}\end{aligned}$$

$\overline{x}$ denotes again the integer variables modified in s. The role of p' is to carry over the permissions p that are gained or lost by the code following the loop, taking into account any state changes performed by the loop. Intuitively, the maximum expressions replace the variables $\overline{x}$ in p with expressions that do not depend on these variables but nonetheless reflect properties of their values right after the execution of the loop. For permissions gained, these properties are based on the under-approximate loop invariant to ensure that they hold for any possible loop execution. For permissions lost, we use the over-approximate invariant. For the loop in `parCopyEven` we use the invariant $0 \leq \texttt{j} \leq \texttt{len(a)}/2$ to obtain $d = -\max_{\texttt{j}|0\leq\texttt{j}<\texttt{len(a)}/2}((q_a{=}\texttt{a} \wedge q_i = 2{*}\texttt{j}\ ?\ 1/2 : 0) + (q_a{=}\texttt{a} \wedge q_i{=}2{*}\texttt{j}{+}1\ ?\ 1 : 0))$. Since there are no statements following the loop, p and therefore p' are 0.

Using the same d term, we can now define the general case of *pre* for loops, combining (1) the loop precondition and (2) the permissions required by the code after the loop, adjusted by the permissions gained or lost during loop execution:

$$\textit{pre}(\texttt{while}\ (b)\ \{\ s\ \},p) = (b\ ?\ \max(\max_{\overline{x}|\mathcal{I}^+ \wedge b} \textit{pre}(s,0)^{\uparrow}, \max_{\overline{x}|\mathcal{I}^+ \wedge \neg b} (p^{\uparrow}) - d) : p)$$

Similarly to p' in the rule for Δ, the expression $\max_{\overline{x}|\mathcal{I}^+ \wedge \neg b}(p^{\uparrow})$ conservatively over-approximates the permissions required to execute the code after the loop.

[1] An under-approximate loop invariant must be true *only* for states that will actually be encountered when executing the loop.

For method `parCopyEven`, we obtain a sufficient precondition that is the negation of the Δ. Consequently, the postcondition is 0.

Soundness. Our *pre* and Δ definitions yield a sound method for computing sufficient permission preconditions and guaranteed postconditions:

Theorem 3 (Soundness of Permission Inference). *For any statement s, if every* `while` *loop in s either is exhale-free or satisfies the condition of Thm. 2 then $pre(s, 0)$ is a sufficient permission precondition for s, and $pre(s, 0) + \Delta(s, 0)$ is a corresponding guaranteed permission postcondition.*

Our inference expresses pre and postconditions using a maximum operator over an unbounded set of values. However, this operator is not supported by SMT solvers. To be able to use the inferred conditions for SMT-based verification, we provide an algorithm for eliminating these operators, as we discuss next.

5 A Maximum Elimination Algorithm

We now present a new algorithm for replacing maximum expressions over an unbounded set of values (called *pointwise maximum expressions* in the following) with equivalent expressions containing no pointwise maximum expressions. Note that, technically our algorithm computes solutions to $\max_{x|b \wedge p \geq 0}(p)$ since some optimisations exploit the fact that the permission expressions our analysis generates always denote non-negative values.

5.1 Background: Quantifier Elimination

Our algorithm builds upon ideas from Cooper's classic *quantifier elimination* algorithm [12] which, given a formula $\exists x.b$ (where b is a quantifier-free Presburger formula), computes an equivalent quantifier-free formula b'. Below, we give a brief summary of Cooper's approach.

The problem is first reduced via boolean and arithmetic manipulations to a formula $\exists x.b$ in which x occurs at most once per literal and with no coefficient. The key idea is then to reduce $\exists x.b$ to a disjunction of two cases: (1) there is a *smallest* value of x making b true, or (2) b is true for *arbitrarily small* values of x.

In case (1), one computes a *finite* set of expressions S (the b_i in [12]) guaranteed to include the smallest value of x. For each (in/dis-)equality literal containing x in b, one collects a *boundary expression* e which denotes a value for x making the literal true, while the value $e - 1$ would make it false. For example, for the literal $y < x$ one generates the expression $y + 1$. If there are no (non-)divisibility constraints in b, by definition, S will include the smallest value of x making b true. To account for (non-)divisibility constraints such as $x\%2{=}0$, the lowest-common-multiple δ of the divisors (and 1) is returned along with S; the guarantee is then that the smallest value of x making b true will be $e + d$ for some $e \in S$ and $d \in [0, \delta - 1]$. We use $\langle\!\langle b \rangle\!\rangle_{small(x)}$ to denote the function handling this computation. Then, $\exists x.b$ can be reduced to $\bigvee_{e \in S, d \in [0, \delta-1]} b[e + d/x]$, where $(S, \delta) = \langle\!\langle b \rangle\!\rangle_{small(x)}$.

In case (2), one can observe that the (in/dis-)equality literals in b will flip value at finitely many values of x, and so for *sufficiently small* values of x, *each* (in/dis-)equality literal in b will have a constant value (e.g., $y > x$ will be true). By replacing these literals with these constant values, one obtains a new expression b' equal to b for small enough x, and which depends on x only via (non-)divisibility constraints. The value of b' will therefore actually be determined by $x \mod \delta$, where δ is the lowest-common-multiple of the (non-)divisibility constraints. We use $\langle\!\langle b \rangle\!\rangle_{-\infty(x)}$ to denote the function handling this computation. Then, $\exists x.b$ can be reduced to $\bigvee_{d\in[0,\delta-1]} b'[d/x]$, where $(b', \delta) = \langle\!\langle b \rangle\!\rangle_{-\infty(x)}$.

In principle, the maximum of a function $y = \max_x f(x)$ can be defined using two first-order quantifiers $\forall x.f(x) \leq y$ and $\exists x.f(x) = y$. One might therefore be tempted to tackle our maximum elimination problem using quantifier elimination directly. We explored this possibility and found two serious drawbacks. First, the resulting formula does not yield a permission-typed expression that we can plug back into our analysis. Second, the resulting formulas are extremely large (e.g., for the `copyEven` example it yields several pages of specifications), and hard to simplify since relevant information is often spread across many terms due to the two separate quantifiers. Our maximum elimination algorithm addresses these drawbacks by natively working with arithmetic expression, while mimicking the basic ideas of Cooper's algorithm and incorporating domain-specific optimisations.

5.2 Maximum Elimination

The first step is to reduce the problem of eliminating general $\max_{x|b}(p)$ terms to those in which b and p come from a simpler restricted grammar. These *simple permission expressions* p do not contain general conditional expressions $(b' \,?\, p_1 : p_2)$, but instead only those of the form $(b' \,?\, r : 0)$ (where r is a constant or rd). Furthermore, simple permission expressions only contain subtractions of the form $p - (b' \,?\, r : 0)$. This is achieved in a precursory rewriting of the input expression by, for instance, distributing pointwise maxima over conditional expressions and binary maxima. For example, the pointwise maximum term (part of the `copyEven` example): $\max_{\mathtt{j}|0\leq \mathtt{j} < \mathtt{len(a)}}((\mathtt{j}\%2{=}0 \,?\, (q_a{=}\mathsf{a} \wedge q_i{=}\mathtt{j} \,?\, \mathsf{rd} : 0) : (q_a{=}\mathsf{a} \wedge q_i{=}\mathtt{j} \,?\, 1 : 0)))$ will be reduced to:

$$\begin{aligned}\max(\ &\max_{\mathtt{j}|0\leq \mathtt{j} < \mathtt{len(a)} \wedge \mathtt{j}\%2=0}((q_a{=}\mathsf{a} \wedge q_i{=}\mathtt{j} \,?\, \mathsf{rd} : 0)),\\ &\max_{\mathtt{j}|0\leq \mathtt{j} < \mathtt{len(a)} \wedge \mathtt{j}\%2\neq 0}((q_a{=}\mathsf{a} \wedge q_i{=}\mathtt{j} \,?\, 1 : 0)))\end{aligned}$$

Arbitrarily-small Values. We exploit a high-level case-split in our algorithm design analogous to Cooper's: given a pointwise maximum expression $\max_{x|b}(p)$, either a *smallest* value of x exists such that p has its maximal value (and b is true), or there are *arbitrarily small* values of x defining this maximal value. To handle the latter case, we define a completely analogous $\langle\!\langle p \rangle\!\rangle_{-\infty(x)}$ function, which recursively replaces all boolean expressions b' in p with $\langle\!\langle b' \rangle\!\rangle_{-\infty(x)}$ as computed by Cooper; we relegate the definition to Sec. B.3 of the TR [**?**]. We then use $(b' \,?\, p' : 0)$, where $(b', \delta_1) = \langle\!\langle b \rangle\!\rangle_{-\infty(x)}$ and $(p', \delta_2) = \langle\!\langle p \rangle\!\rangle_{-\infty(x)}$, as our expression in this case.

$$\begin{aligned}
\langle\!\langle (b\,?\,p:0)\rangle\!\rangle_{smallmax(x)} &= (T,\delta),\ \textit{where}\ (S,\delta) = \langle\!\langle b\rangle\!\rangle_{small(x)}, T = \{(e,\mathsf{true}) \mid e \in S\}\\
\langle\!\langle p_1 + p_2\rangle\!\rangle_{smallmax(x)} &= (T_1 \cup T_2, lcm(\delta_1,\delta_2))\\
\textit{where}\quad (T_1,\delta_1) &= \langle\!\langle p_1\rangle\!\rangle_{smallmax(x)},\ (T_2,\delta_2) = \langle\!\langle p_2\rangle\!\rangle_{smallmax(x)}\\
\langle\!\langle \max(p_1,p_2)\rangle\!\rangle_{smallmax(x)} &= \langle\!\langle \min(p_1,p_2)\rangle\!\rangle_{smallmax(x)} = \langle\!\langle p_1 + p_2\rangle\!\rangle_{smallmax(x)}\ \textit{as above}\\
\langle\!\langle p_1 - (b\,?\,p:0)\rangle\!\rangle_{smallmax(x)} &= (T_1 \cup T_2, lcm(\delta_1,\delta_2))\\
\textit{where}\quad (T_1,\delta_1) &= \langle\!\langle p_1\rangle\!\rangle_{smallmax(x)},\ (S_2,\delta_2) = \langle\!\langle \neg b\rangle\!\rangle_{small(x)},\\
T_2' &= \{(e, p_1 > 0) \mid\ e \in S_2\}
\end{aligned}$$

$$\begin{aligned}
&\langle\!\langle (p,b)\rangle\!\rangle_{smallmax(x)} = (T_p \cup T_b', \delta')\ \textit{where}\ (T_p,\delta_p) = \langle\!\langle p\rangle\!\rangle_{smallmax(x)},\ (S_b,\delta_b) = \langle\!\langle b\rangle\!\rangle_{small(x)},\\
&\delta' = lcm(\delta_p,\delta_b),\ (b',\delta_b) = \langle\!\langle b\rangle\!\rangle_{-\infty(x)},\ (p',\delta_p) = \langle\!\langle p\rangle\!\rangle_{-\infty(x)},\\
&T_b' = \{(e_b, (\bigvee_{d\in[0,\delta'-1]} ((\neg b' \wedge p' > 0)[d/x])) \vee \bigvee_{\substack{(e_p,b_p)\in T_p\\ d_p\in[0,\delta_p-1]}} (\neg b \wedge b_p)[(e_p + d_p)/x])\ \mid\ e_b \in S_b\}
\end{aligned}$$

Fig. 5: Filtered boundary expression computation.

Note that this expression still depends on x if it contains (non-)divisibility constraints; Thm. 4 shows how x can be eliminated using δ_1 and δ_2.

Selecting Boundary Expressions for Maximum Elimination. Next, we consider the case of selecting an appropriate set of boundary expressions, given a $\max_{x|b}(p)$ term. We define this first for p in isolation, and then give an extended definition accounting for the b. Just as for Cooper's algorithm, the boundary expressions must be a set guaranteed to include the *smallest* value of x defining the maximum value in question. The set must be finite, and be as small as possible for efficiency of our overall algorithm. We refine the notion of boundary expression, and compute a set of *pairs* (e, b') of integer expression e and its *filter condition* b': the filter condition represents an additional condition under which e must be included as a boundary expression. In particular, in contexts where b' is false, e can be ignored; this gives us a way to symbolically define an ultimately-smaller set of boundary expressions, particularly in the absence of contextual information which might later show b' to be false. We call these pairs *filtered boundary expressions.*

Definition 1 (Filtered Boundary Expressions). *The* filtered boundary expression computation for x in p, *written* $\langle\!\langle p\rangle\!\rangle_{smallmax(x)}$, *returns a pair of a set* T *of pairs* (e, b'), *and an integer constant* δ, *as defined in Fig. 5. This definition is also overloaded with a definition of* filtered boundary expression computation for $(x \mid b)$ in p, *written* $\langle\!\langle (p,b)\rangle\!\rangle_{smallmax(x)}$.

Just as for Cooper's $\langle\!\langle b\rangle\!\rangle_{small(x)}$ computation, our function $\langle\!\langle p\rangle\!\rangle_{smallmax(x)}$ computes the set T of (e, b') pairs along with a single integer constant δ, which is the least common multiple of the divisors occurring in p; the desired smallest value of x may actually be some $e + d$ where $d \in [0, \delta - 1]$. There are three key points to Def. 1 which ultimately make our algorithm efficient:

First, the case for $\langle\!\langle (b\,?\,p:0)\rangle\!\rangle_{smallmax(x)}$ only includes boundary expressions for making b *true*. The case of b being false (from the structure of the permission expression) is not relevant for trying to maximise the permission expression's value (note that this case will never apply under a subtraction operator, due to our simplified grammar, and the case for subtraction not recursing into the right-hand operand).

Second, the case for $\langle\!\langle p_1 - (b\,?\,p:0)\rangle\!\rangle_{smallmax(x)}$ dually only considers boundary expressions for making b *false* (along with the boundary expressions for maximising p_1). The filter condition $p_1 > 0$ is used to drop the boundary expressions for making b false; in case p_1 is not strictly positive we know that the evaluation of the whole permission expression will not yield a strictly-positive value, and hence is not an interesting boundary value for a non-negative maximum.

Third, in the overloaded definition of $\langle\!\langle (p,b)\rangle\!\rangle_{smallmax(x)}$, we combine boundary expressions for p with those for b. The boundary expressions for b are, however, superfluous *if*, in analysing p we have already determined a value for x which maximises p and happens to satisfy b. If all boundary expressions for p (whose filter conditions are true) make b true, *and* all non-trivial (i.e. strictly positive) evaluations of $\langle\!\langle p\rangle\!\rangle_{-\infty(x)}$ used for potentially defining p's maximum value also satisfy b, then we can safely discard the boundary expressions for b.

We are now ready to reduce pointwise maximum expressions to equivalent maximum expressions over finitely-many cases:

Theorem 4 (Simple Maximum Expression Elimination). *For any pair* (p,b)*, if* $\models p \geq 0$*, then we have:*

$$\models \max_{x|b} p = \max\Big(\max_{\substack{(e,b'')\in T\\ d\in[0,\delta-1]}} (b'' \wedge b[e+d/x]\,?\,p[e+d/x]:0)),\\ \max_{d\in[0,lcm(\delta_1,\delta_2)-1]} (b'[d/x]\,?\,p'[d/x]:0)\Big)$$

where $(T,\delta) = \langle\!\langle (p,b)\rangle\!\rangle_{smallmax(x)}$, $(b',\delta_1) = \langle\!\langle b\rangle\!\rangle_{-\infty(x)}$ *and* $(p',\delta_2) = \langle\!\langle p\rangle\!\rangle_{-\infty(x)}$.

To see how our filter conditions help to keep the set T (and therefore, the first iterated maximum on the right of the equality in the above theorem) small, consider the example: $\max_{x|x\geq 0}((x{=}i\,?\,1:0))$ (so p is $(x{=}i?1{:}0)$, while b is $x \geq 0$). In this case, evaluating $\langle\!\langle (p,b)\rangle\!\rangle_{smallmax(x)}$ yields the set $T = \{(i,\mathsf{true}),(0,i<0)\}$ with the meaning that the boundary expression i is considered in all cases, while the boundary expression 0 is only of interest if $i < 0$. The first iterated maximum term would be $\max((\mathsf{true}\wedge i{\geq}0\,?\,(i{=}i\,?\,1:0):0),(i{<}0\wedge 0{\geq}0\,?\,(0{=}i\,?\,1:0):0))$. We observe that the term corresponding to the boundary value 0 can be simplified to 0 since it contains the two contradictory conditions $i < 0$ and $0 = i$. Thus, the entire maximum can be simplified to $(i{\geq}0\,?\,1:0)$. Without the filter conditions the result would instead be $\max((i{\geq}0\,?\,1:0),(0{=}i\,?\,1:0))$. In the context of our permission analysis, the filter conditions allow us to avoid generating boundary expressions corresponding e.g. to the integer loop invariants, provided that the expressions generated by analysing the permission expression in question already suffice. We employ aggressive syntactic simplification of the resulting expressions, in order to exploit these filter conditions to produce succinct final answers.

Program	LOC	Loops	Size	Prec.	Time
addLast	12	1 (1)	1.9	✓	21
append	13	1 (1)	1.9	✓	32
array1	17	2 (2)	0.9	✗	28
array2	23	3 (2)	0.9	✗	35
array3	23	2 (2)	1.1	✓	24
arrayRev	18	1 (1)	3.2	✓*	28
bubbleSort	23	2 (2)	1.8	✓*	34
copy	16	2 (1)	1.6	✓	27
copyEven	17	1 (1)	1.6	✓	27
copyEven2	14	1 (1)	1.4	✗	20
copyEven3	14	1 (1)	2.2	✓*	23
copyOdd	21	2 (1)	2.4	✓	55
copyOddBug	19	2 (1)	7.1	✓	57
copyPart	17	2 (1)	1.7	✓	30
countDown	21	3 (2)	1.1	✓	32
diff	31	2 (2)	2.0	✗	70
find	19	1 (1)	3.0	✓	43
findNonNull	19	1 (1)	3.0	✓	40
init	18	2 (1)	1.1	✓	28
init2d	23	2 (2)	2.1	✓	52
initEven	18	2 (1)	0.9	✗	26
initEvenbug	18	2 (1)	1.5	✗	28
initNonCnst	18	2 (1)	1.1	✓	27
initPart	19	2 (1)	1.1	✓	30

Program	LOC	Loops	Size	Prec.	Time
initPartBug	19	2 (1)	1.5	✓	31
insertSort	21	2 (2)	2.5	✓*	35
javaBubble	24	2 (2)	2.3	✓*	32
knapsack	21	2 (2)	1.3	✗	45
lis	37	4 (2)	4.2	✓	73
matrixmult	33	3 (3)	1.5	✓	78
mergeinter	23	2 (1)	3.4	✗	56
mergeintbug	23	2 (1)	2.6	✗	59
memcopy	16	2 (1)	1.6	✓	28
multarray	26	2 (2)	2.1	✓	40
parcopy	20	2 (1)	1.2	✓	30
pararray	20	2 (1)	1.2	✓	31
parCopyEven	22	2 (1)	5.0	✓*	79
parMatrix	35	4 (2)	1.1	✓	80
parNested	31	4 (2)	0.5	✗	57
relax	33	1 (1)	1.4	✓*	55
reverse	21	2 (1)	3.9	✓	42
reverseBug	21	2 (1)	1.7	✓	42
sanfoundry	27	2 (1)	2.1	✓	37
selectSort	26	2 (2)	1.0	✗	38
strCopy	16	2 (1)	0.9	✗	21
strLen	10	1 (1)	0.8	✗	15
swap	15	1 (1)	1.5	✓	19
swapBug	15	1 (1)	1.5	✓	19

Table 1: Experimental results. For each program, we list the lines of code and the number of loops (in brackets the nesting depth). We report the relative size of the inferred specifications compared to hand-written specifications, and whether the inferred specifications are precise (a star next to the tick indicates slightly more precise than hand-written specifications). Inference times are given in ms.

6 Implementation and Experimental Evaluation

We have developed a prototype implementation of our permission inference. The tool is written in Scala and accepts programs written in the Viper language [38], which provides all the features needed for our purposes.

Given a Viper program, the tool first performs a forward numerical analysis to infer the over-approximate loop invariants needed for our handling of loops. The implementation is parametric in the numerical abstract domain used for the analysis; we currently support the abstract domains provided by the Apron library [24]. As we have yet to integrate the implementation of under-approximate invariants (e.g., [35]), we rely on user-provided invariants, or assume them to be false if none are provided. In a second step, our tool performs the inference and

maximum elimination. Finally, it annotates the input program with the inferred specification.

We evaluated our implementation on 43 programs taken from various sources; included are all programs that do not contain strings from the array memory safety category of SV-COMP 2017, all programs from Dillig et al. [15] (except three examples involving arrays of arrays), loop parallelisation examples from VerCors [5], and a few programs that we crafted ourselves. We manually checked that our soundness condition holds for all considered programs. The parallel loop examples were encoded as two consecutive loops where the first one models the forking of one thread per loop iteration (by iteratively exhaling the permissions required for all loop iterations), and the second one models the joining of all these threads (by inhaling the permissions that are left after each loop iteration). For the numerical analysis we used the *polyhedra abstract domain* provided by Apron. The experiments were performed on a dual core machine with a 2.60 GHz Intel Core i7-6600U CPU, running Ubuntu 16.04.

An overview of the results is given in Table 1. For each program, we compared the size and precision of the inferred specification with respect to hand-written ones. The running times were measured by first running the analysis 50 times to warm up the JVM and then computing the average time needed over the next 100 runs. The results show that the inference is very efficient. The inferred specifications are concise for the vast majority of the examples. In 35 out of 48 cases, our inference inferred precise specifications. Most of the imprecisions are due to the inferred numerical loop invariants. In all cases, manually strengthening the invariants yields a precise specification. In one example, the source of imprecision is our abstraction of array-dependent conditions (see Sec. 4).

7 Related Work

Much work is dedicated to the analysis of array programs, but most of it focuses on array content, whereas we infer permission specifications. The simplest approach consists of "smashing" all array elements into a single memory location [4]. This is generally quite imprecise, as only weak updates can be performed on the smashed array. A simple alternative is to consider array elements as distinct variables [4], which is feasible only when the length of the array is statically-known. More-advanced approaches perform syntax-based [18, 22, 25] or semantics-based [13, 34] partitions of an array into symbolic segments. These require segments to be contiguous (with the exception of [34]), and do not easily generalise to multidimensional arrays, unlike our approach. Gulwani et al. [20] propose an approach for inferring quantified invariants for arrays by lifting quantifier-free abstract domains. Their technique requires templates for the invariants.

Dillig et al. [15] avoid an explicit array partitioning by maintaining constraints that over- and under-approximate the array elements being updated by a program statement. Their work employs a technique for directly generalising the analysis of a single loop iteration (based on quantifier elimination), which works well when different loop iterations write to disjoint array locations. Gedell and Hähnle

[17] provide an analysis which uses a similar criterion to determine that it is safe to parallelise a loop, and treat its heap updates as one bulk effect. The condition for our projection over loop iterations is weaker, since it allows the same array location to be updated in multiple loop iterations (like for example in sorting algorithms). Blom et al. [5] provide a specification technique for a variety of parallel loop constructs; our work can infer the specifications which their technique requires to be provided.

Another alternative for generalising the effect of a loop iteration is to use a first order theorem prover as proposed by Kovács and Voronkov [28]. In their work, however, they did not consider nested loops or multidimensional arrays. Other works rely on loop acceleration techniques [1, 7]. In particular, like ours, the work of Bozga et al. [7] does not synthesise loop invariants; they directly infer post-conditions of loops with respect to given preconditions, while we additionally infer the preconditions. The acceleration technique proposed in [1] is used for the verification of array programs in the tool Booster [2].

Monniaux and Gonnord [36] describe an approach for the verification of array programs via a transformation to array-free Horn clauses. Chakraborty et al. [11] use heuristics to determine the array accesses performed by a loop iteration and split the verification of an array invariant accordingly. Their non-interference condition between loop iterations is similar to, but stronger than our soundness condition (cf. Sec. 4). Neither work is concerned with specification inference.

A wide range of static/shape analyses employ tailored separation logics as abstract domain (e.g., [3, 19, 10, 29, 41]); these works handle recursively-defined data structures such as linked lists and trees, but not random-access data structures such as arrays and matrices. Of these, Gulavani et al. [19] is perhaps closest to our work: they employ an integer-indexed domain for describing recursive data structures. It would be interesting to combine our work with such separation logic shape analyses. The problems of automating biabduction and entailment checking for array-based separation logics have been recently studied by Brotherston et al. [8] and Kimura et al. [27], but have not yet been extended to handle loop-based or recursive programs.

8 Conclusion and Future Work

We presented a precise and efficient permission inference for array programs. Although our inferred specifications contain redundancies in some cases, they are human readable. Our approach integrates well with permission-based inference for other data structures and with permission-based program verification.

As future work, we plan to use SMT solving to further simplify our inferred specifications, to support arrays of arrays, and to extend our work to an interprocedural analysis and explore its combination with biabduction techniques.

Acknowledgements. We thank Seraiah Walter for his earlier work on this topic, and Malte Schwerhoff and the anonymous reviewers for their comments and suggestions. This work was supported by the Swiss National Science Foundation.

References

1. F. Alberti, S. Ghilardi, and N. Sharygina. Definability of Accelerated Relations in a Theory of Arrays and Its Applications. In *FroCoS*, pages 23–39, 2013.
2. F. Alberti, S. Ghilardi, and N. Sharygina. Booster: An Acceleration-Based Verification Framework for Array Programs. In *ATVA*, pages 18–23, 2014.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*, 2005.
4. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. In *AIAA*, 2010.
5. S. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In *FASE*, pages 202–217, 2015.
6. J. Boyland. Checking Interference with Fractional Permissions. In *SAS 2003*, volume 2694 of *LNCS*, pages 55–72, 2003.
7. M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic Verification of Integer Array Programs. In *CAV*, pages 157–172, 2009.
8. J. Brotherston, N. Gorogiannis, and M. Kanovich. Biabduction (and related problems) in array separation logic. In *Proceedings of CADE-26*, volume 10395 of *LNAI*, pages 472–490. Springer, 2017.
9. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.
10. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, Dec. 2011.
11. S. Chakraborty, A. Gupta, and D. Unadkat. Verifying Array Manipulating Programs by Tiling. In *SAS*, pages 428–449, 2017.
12. D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine intelligence*, 7(91-99):300, 1972.
13. P. Cousot, R. Cousot, and F. Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *POPL*, pages 105–118, 2011.
14. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
15. I. Dillig, T. Dillig, and A. Aiken. Fluid Updates: Beyond Strong vs. Weak Updates. In *ESOP*, pages 246–266, 2010.
16. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *International Symposium on Programming*, pages 125–132, 1984.
17. T. Gedell and R. Hähnle. Automating Verification of Loops by Parallelization. In *LPAR*, pages 332–346, 2006.
18. D. Gopan, T. W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Array Operations. In *POPL*, pages 338–350, 2005.
19. B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-Up Shape Analysis. In *SAS*, pages 188–204, 2009.
20. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *POPL*, pages 235–246, 2008.
21. A. Gupta and A. Rybalchenko. InvGen: An Efficient Invariant Generator. In *CAV*, pages 634–640, 2009.
22. N. Halbwachs and M. Péron. Discovering Properties About Arrays in Simple Programs. In *PLDI*, pages 339–348, 2008.

23. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55, 2011.
24. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.
25. R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *CAV*, pages 193–206, 2007.
26. N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August. A collaborative dependence analysis framework. In *CGO*, pages 148–159, 2017.
27. D. Kimura and M. Tatsuta. Decision Procedure for Entailment of Symbolic Heaps with Arrays. In *APLAS*, pages 169–189, 2017.
28. L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *FASE*, pages 470–485, 2009.
29. Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape analysis via second-order bi-abduction. In *CAV*, pages 52–68, 2014.
30. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
31. K. R. M. Leino and P. Müller. A Basis for Verifying Multi-Threaded Programs. In *ESOP*, volume 5502 of *LNCS*, pages 378–393, 2009.
32. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426, 2010.
33. S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *POPL*, pages 270–282, 2002.
34. J. Liu and X. Rival. An Array Content Static Analysis Based on Non-Contiguous Partitions. *Computer Languages, Systems & Structures*, 47:104–129, 2017.
35. A. Miné. Inferring Sufficient Conditions with Backward Polyhedral Under-Approximations. *Electronic Notes in Theoretical Computer Science*, 287:89–100, 2012.
36. D. Monniaux and L. Gonnord. Cell Morphing: From Array Programs to Array-Free Horn Clauses. In *SAS*, pages 361–382, 2016.
37. P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.
38. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.
39. R. Piskac, T. Wies, and D. Zufferey. Grasshopper – complete heap verification with mixed specifications. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
40. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
41. R. N. S. Rowe and J. Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 53–65, New York, NY, USA, 2017. ACM.
42. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In *VMCAI*, pages 199–215, 2005.

43. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
44. A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In *TACAS*, 2018.
45. J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC-FSE)*, pages 205–214. ACM, 2007.