EPiC Series in Computing

Volume 46, 2017, Pages 368–384

LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Quantified Heap Invariants for Object-Oriented Programs

Temesghen Kahsai[1,4], Rody Kersten[1], Philipp Rümmer[2], and Martin Schäf[3]

[1] Carnegie Mellon University, Silicon Valley
[2] Uppsala University
[3] SRI International
[4] NASA Ames

Abstract

Heap and data structures represent one of the biggest challenges when applying model checking to the analysis of software programs: in order to verify (unbounded) safety of a program, it is typically necessary to formulate quantified inductive invariants that state properties about an unbounded number of heap locations. Methods like Craig interpolation, which are commonly used to infer invariants in model checking, are often ineffective when a heap is involved. To address this challenge, we introduce a set of new proof and program transformation rules for verifying object-oriented programs with the help of *space invariants,* which (implicitly) give rise to quantified invariants. Leveraging advances in Horn solving, we show how space invariants can be derived fully automatically, and how the framework can be used to effectively verify safety of Java programs.

1 Introduction

One of the challenging areas for software verification is automatic reasoning about heap data. In order to be fully precise, one needs to model heap manipulation in an expressive logic that includes, for example, the theory of arrays with quantifiers [5] where reasoning is undecidable and support for common abstraction techniques, such as Craig interpolation, is limited.

A sound alternative is to abstract the effect of heap interactions using invariants that capture properties of relevant heap regions. For type-safe, object-oriented languages, like Java, this requires finding invariants that summarize the possible states of all objects that a particular reference might point to at a given program point. For instance, in the setting of deductive verification, it is common practice to use annotations like *object invariants* [22] and *class invariants* [19] to capture all possible states of objects of a given type throughout its lifetime, including both object and static fields. While these approaches are powerful and can be used to verify complex behavioral properties of programs, invariants are in general hard to infer automatically: it can frequently be necessary to find invariants that span multiple objects, or whole data-structures, and often additional information like specifications of library methods is needed before program safety can be shown.

In this paper, we propose the concept of *space invariants,*[1] a lightweight notion of invariant inspired by frameworks such as *refinement types* and *liquid types* (e.g., [1, 11, 16, 28, 31]). Our

[1]The name is derived from the fact that space invariants can capture both data and shape properties of heap data-structures.

motivation is to automatically find invariants to summarize the states of an object, specific to one particular program and a set of properties to be proven. The approach is also motivated by the success of model checkers in finding loop invariants, where the inferred invariants are just sufficient to verify the program and, by no means, need to precisely describe the program states reachable in the loop.

In contrast to class or object invariants, as commonly used in deductive verification systems, a particular feature of space invariants is the ability to capture *context-sensitive* information: space invariants are able to focus on the subset of objects that can be referenced by a variable at a certain program point, and space invariants can distinguish different object states based on the current control state of a program.

We propose a simple but effective solution for automatically computing such space invariants, leveraging recent advances in Horn solving technology. The presented technique is part of a verification algorithm for single-threaded Java programs and is implemented in the JayHorn verification tool [15].

To obtain space invariants, we first simplify the heap interaction of programs by a set of program transformations that re-arrange heap read and write statements, in a way that allows us to read or write as many fields of an object as possible in a single transaction (instead of accessing these fields one-by-one). We refer to these transactions as pull (to read all fields of one object into local variables) and push (to update all fields of an object on the heap simultaneously). We show how programs can be transformed to use pull and push, and also provide a simplification step to eliminate unnecessary heap interactions.

Heap interactions in the resulting program can be abstracted using (symbolic) space invariants: a space invariant is *assumed* when fields of an object are read (pulled), and *asserted* when fields are updated (pushed). After replacing pull and push by space invariants, our program does not have any heap interaction and can be passed to a Horn clause solver like Spacer [17] or Eldarica [29], which will then try to instantiate the symbolic space invariants with concrete formulas to verify the assertions encoded in the original Java program.

The contributions of the paper are: (1) a new automatic verification paradigm for object-oriented programs using space invariants and a translation to Horn constraints; (2) supporting this paradigm, a set of program transformations for restructuring and optimizing heap access in object-oriented programs, and a number of extensions to tune the precision of the verification methodology; (3) experimental evaluation using Java benchmarks from different sources.

2 Verification Example

We demonstrate how space invariants can be used to verify safety properties of Java programs that allocate an unbounded amount of memory. Figure 1 shows a program that will serve as our running example; while the example itself is contrived, we believe that it illustrates a realistic scenario of deriving properties about unbounded data-structures. The main method allocates an array `table` in line 13. From line 16 to 23, it generates two lists, `l1` and `l2`, of `Node` objects. It adds all `args` with a value between zero and `size` to `l1`, and all other `args` to `l2`. From line 24 to 27, the main method iterates over all elements in `l1` and accesses the array `table` at a position determined by the `data` field of the current node in `l1` (line 24).

In the following, we show how we can use space invariants to prove that the array access in line 25 is always within bounds.[2] Verifying the correctness of the array access in line 25 is

[2]Note that the program may still throw an exception if `args` contains an element that cannot be parsed as an integer, in which case `Integer.parseInt` in line 17 throws an exception; for this example we are only interested in the safety of the array access in line 25.

```
 1 public static class Node {
 2   final Node next;
 3   final int data;
 4
 5   public Node(Node next, int data) {
 6     this.next = next;
 7     this.data = data;
 8   }
 9 }
10
11 public static void main(String[] args) {
12   final int size = 10;
13   final int[] table = new int[size] ;
14   Node l1 = null;
15   Node l2 = null;
16   for (int i=0; i<args.length; i++) {
17       int d = Integer.parseInt(args[i]);
18       if (d >= 0 && d < size) {
19         l1 = new Node(l1, d);
20       } else {
21         l2 = new Node(l2, d);
22       }
23   }
24   while (l1 != null) {
25     table[l1.data] = table[l1.data] + 1;
26     l1 = l1.next;
27   }
28 }
```

Figure 1: Running example to illustrate our encoding of Java and how space invariants help to prove safety properties of Java programs. The main method builds up two lists of Node objects (line 16-23) and then increments elements in an array at positions determined by the data in the first list. Our goal is to prove that the array access in line 25 is always within bounds.

challenging for automated tools, since it requires combined reasoning about the shape of heap-allocated linked data-structures, and about arithmetic properties of the stored data. Indeed, to the best of our knowledge *no fully-automatic tool is able to verify the program in Figure 1,* with the exception of our own JayHorn verifier [15].

To verify the program, intuitively we need to find an invariant that captures the property that any allocated Node object that can be referenced by the variable `l1` has the property that $0 \leq \texttt{l1.data} < 10$. A naïve choice would be an invariant of the form:

$$\forall (o : \texttt{Node}).\ 0 \leq o.data < 10$$

Unfortunately, this invariant does not hold for the objects in list `l2`: we need to be able to distinguish between objects occurring in `l1` and `l2`. This could, for instance, be done by adding a reachability assumption to the invariant:

$$\forall (o : \texttt{Node}).\ (\texttt{l1} \xrightarrow{*} o) \rightarrow 0 \leq o.data < 10$$

where $\texttt{l1} \xrightarrow{*} o$ expresses that o occurs in list `l1`. Due to its expressiveness, the resulting logic is difficult to handle in a fully-automated model checker, however, and useful primarily for interactive verification (e.g., as in [26]).

We can instead make the simpler observation that objects can also be classified based on the *allocation site*, since objects for list `l1` will always be created in line 19, objects for `l2` always in line 21. To be able to make this classification, we add an additional field to each class:

```
final int allocSite;
```

that is assigned during object construction to the location of the `new` statement that creates the object (we enumerate all `new` statements in the program, and pass the number associated with a `new` as additional argument to the constructor). With this field, we can express a more specific invariant about objects of type `Node`:

$$\forall (o : \texttt{Node}).\ \ o.\mathit{allocSite} = \texttt{l19} \rightarrow \big(0 \leq o.\mathit{data} < 10 \wedge (o.\mathit{next} = \texttt{null} \vee o.\mathit{next}.\mathit{allocSite} = \texttt{l19})\big)$$

The invariant states that, if an object of type `Node` has been allocated at line 19 then the value of `data` is between zero and ten, and the object referenced by `next` has also been allocated at line 19 or is `null`. This invariant holds for all `Node` objects on the heap and is sufficient to verify that the array access is safe.

The important thing to note is that this invariant does not need to mention `l2` or any other object on the heap. All we need to know can be expressed using simply the allocation site variable. *Automatically inferring invariants like this, which hold for all allocated objects of a given class, is the main goal of this paper.* Apart from the allocation site, space invariants can take various other (immutable) features of objects into account, as well as results from standard static analysis techniques (Section 3.2). We show that with this added information, space invariants can be precise enough to construct proofs of realistic programs, while also simple enough to be inferred automatically by off-the-shelf Horn solvers.

Inferring space invariants. We show how we can infer such space invariants automatically with our JayHorn tool [15]. The tool and all examples from this paper are available online.[3]

Before we can find the space invariant for this example, we have to transform the program a little. This is done as part of the translation into our intermediate verification language (IVL), which we describe in detail in Section 3.1. These transformations include replacing exceptional flow by conditional choices and helper variables (e.g., methods return pairs of return value and thrown exception, and callers have to check if this exception is non-null before using the return value) and making dispatch of virtual calls explicit by adding switch cases over the dynamic type of objects. Further, all methods are transformed into static methods (taking the *this* pointer as first argument), and all fields are made public (essentially turning them into structs). The resulting program looks a lot like type-safe C code. During that transformation, we also add the `allocSite` field to each class. Since `allocSite` is final, we also add an additional parameter to each constructor that takes an integer to initialize this field. Then, each constructor call gets changed to pass an integer to the constructor that uniquely identifies the `new` statement. For example, line 19 in Figure 1 will be changed to `l1 = new Node(l1, d, 19)` and line 21 will be changed to `l2 = new Node(l2, d, 21)`.

Since the goal of space invariants is to abstract heap behavior, we want to transform the program in a way that minimizes the points of heap interaction to avoid unnecessary loss of precision. For example, without minimization of heap interaction, in line 25 we read `l1.data` twice (once for each look-up in the array), and in line 26 we read `l1.next`. When working

[3] https://github.com/jayhorn/jayhorn/releases/tag/v0.5.1

with space invariants, we can assume that the space invariant holds before every heap read access, and we have to verify that the invariant is preserved by every write access. This would significantly increase the size of our verification conditions: for the two lines, six statements have to be added (two for the reads of `data`, one for the write of `data`, two for read and write of `table`, and one for the read of `next`).

Instead, we introduce two new statements: `pull` to read all fields of an object into local variables, and `push` to update all fields of an object with data stored in local variables. With these statements, lines 25 and 26 can be rewritten as:

```
while  l1 ≠ null
     data, next, allocSite = pull(l1)
     assert(0<=data<10)
     tmp, allocSite = pull(table, data)
     push(table, data, tmp+1, allocSite)
     l1 = next
```

With this encoding, we have reduced the points where we need to invoke the heap from six to three. As we show in Section 4, we can now abstract the heap by replacing all `pull` and `push` statements by symbolic space invariants. For each class, we introduce a predicate for which the arity corresponds to the number of fields of that class. For the class Node, we create the symbolic invariant $\phi_{\texttt{Node}}(\mathit{this}, \mathit{data}, \mathit{next}, \mathit{allocSite})$. For the int-array, we create $\phi_{\texttt{int[]}}(\mathit{this}, \mathit{idx}, \mathit{val}, \mathit{allocSite})$, where idx refers to the index for this array read/write. The index is added for the same reason as the allocation site: to allow differentiation between array cells where possible. A `pull` or `push` statement can then be translated to assumption or assertion, respectively, of the space invariant.

With these symbolic invariants, we can rewrite the snippet above, as follows:

```
while  l1 ≠ null
     havoc(data,tmp,next,allocSite1,allocSite2)
     assume $\phi_{\texttt{Node}}$(l1, data, next, allocSite1)
     assert(0<=data<10)
     assume $\phi_{\texttt{int[]}}$(table, data, tmp, allocSite2)
     assert $\phi_{\texttt{int[]}}$(table, data, tmp+1, allocSite2)
     l1 = next
```

The transformation added a `havoc` statement that assigns non-deterministic values to all its arguments. We need this non-determinism because we have to make sure that the assumption resulting from a `pull` reasons about fresh values in consecutive loop iterations.

After this transformation, we have abandoned our concrete semantics of the original Java program and replaced it by an abstract semantics that relies on space invariants. Since we already made method calls static and all fields public, the resulting program is a simple imperative program that only uses local variables and invariants. The translation of such a program into Horn logic is mostly standard. Now we can employ an off-the-shelf Horn solver to find our space invariants by searching for assignments to $\phi_{\texttt{Node}}$ and $\phi_{\texttt{int[]}}$.

Encoding of References as Tuples. As shown in the running example, it is important to identify distinguishing features of objects to precisely capture their properties using space invariants. Such features can, for example, include the dynamic type of an object (that is, the type it has on the heap, not the type of the current reference), the values of final fields, and, in particular, the value of the `allocSite` variable that we just introduced. Note that such features are *immutable* (i.e., they do not change after object creation), and that they cannot be read via field access (i.e., by assuming the space invariant), since this would defeat the purpose of distinguishing different object contexts: e.g., in line 25 of Figure 1 it has to be known that the reference `l1` points to an object with allocation site `l19`.

For this reason, we encode references using tuple types. A reference is represented by a tuple $(id, type, f_0, ...f_n)$, where id is an integer representing the address on the heap[4], $type$ is the dynamic type of the object (which can be a subtype of the declared type), followed by all immutable (or final) fields. This includes our `allocSite` field and the `length` field of the array `table`. We have to make one exception in the tuple encoding: the `next` field of `Node` cannot be encoded as part of the tuple because it would recursively inline all other objects. That is, we do not add fields to the tuple if their type definition is recursive. These fields will still be accessed using `pull` and `push`, even if they are final.

With the tuple encoding, the variable `l1` is of type $Int \times Type \times Int \times Int$, where the first component represents the id, the second represents the type, the third the value of `allocSite`, and the fourth the value of `data`. In case `l1` points to `null`, the id is 0. For readability, we use the field name as identifier for accessing tuple elements. For example, the fourth element of `l1` will be written as $\mathtt{l1}{\downarrow_{\mathtt{data}}}$.

Note that we can now shorten $\phi_{\mathtt{Node}}(this, data, next, allocSite)$ to $\phi_{\mathtt{Node}}(this, data, next)$ and $\phi_{\mathtt{int[]}}(this, idx, val, allocSite)$ to $\phi_{\mathtt{int[]}}(this, idx, val)$, as the allocation site can be accessed through the tuple (i.e. with $this \downarrow_{\mathtt{allocSite}}$). The statements from above can be encoded as Horn clauses as follows:

$$
\begin{aligned}
p0(\mathtt{l1}, \mathtt{table}) \wedge \mathtt{l1}{\downarrow_{\mathtt{id}}} \neq 0 &\rightarrow p1(\mathtt{l1}, \mathtt{table}) \\
\phi_{\mathtt{Node}}(\mathtt{l1}, \mathtt{l1}{\downarrow_{\mathtt{data}}}, \mathtt{next}) \wedge p1(\mathtt{l1}, \mathtt{table}) &\rightarrow p2(\mathtt{l1}, \mathtt{table}, \mathtt{next}) \\
p2(\mathtt{l1}, \mathtt{table}, \mathtt{next}) &\rightarrow 0 \leq \mathtt{l1}{\downarrow_{\mathtt{data}}} < 10 \\
\phi_{\mathtt{int[]}}(\mathtt{table}, \mathtt{l1}{\downarrow_{\mathtt{data}}}, \mathtt{tmp}) \wedge p2(\mathtt{l1}, \mathtt{table}, \mathtt{next}) &\rightarrow p3(\mathtt{l1}, \mathtt{table}, \mathtt{next}, \mathtt{tmp}) \\
p3(\mathtt{l1}, \mathtt{table}, \mathtt{next}, \mathtt{tmp}) &\rightarrow \phi_{\mathtt{int[]}}(\mathtt{table}, \mathtt{l1}{\downarrow_{\mathtt{data}}}, \mathtt{tmp+1}) \\
p3(\mathtt{l1}, \mathtt{table}, \mathtt{next}, \mathtt{tmp}) &\rightarrow p0(\mathtt{next}, \mathtt{table})
\end{aligned}
$$

We can see that, with this encoding of the loop, if $p1(\mathtt{l1}, \mathtt{table})$ implies that $\mathtt{l1}{\downarrow_{\mathtt{allocSite}}} = \mathtt{l19}$, then we can verify the assertion with the following space invariants:

$$
\begin{aligned}
&\phi_{\mathtt{int[]}}(\mathtt{o}, \mathtt{idx}, \mathtt{data}) := true \\
&\phi_{\mathtt{Node}}(\mathtt{o}, \mathtt{data}, \mathtt{next}) := \mathtt{o}{\downarrow_{\mathtt{allocSite}}} = \mathtt{l19} \rightarrow \big(0 \leq \mathtt{data} < 10 \wedge (\mathtt{next}{\downarrow_{\mathtt{id}}} = 0 \\
&\qquad\qquad \vee\, \mathtt{next}{\downarrow_{\mathtt{allocSite}}} = \mathtt{l19})\big)
\end{aligned}
$$

We omitted several details in this example for the sake of readability, such as the encoding of constructors and how it ensures that the allocation site field is correctly set, or method calls in general, and other more advanced problems like space invariants in combination with sub-typing. In the following, we formalize our IVL and describe the most relevant parts of the translation from Java into this IVL. For more detailed explanations and some background on how design decisions were made, we also refer to the development blog.[5]

[4] It does not have to be the real address. Any numbering of heap locations that ensures that non-aliased objects are represented by different numbers does the job.

[5] http://jayhorn.github.io/jayhorn/jekyll/2016/08/01/model-checking-java/

$$
\begin{aligned}
\mathit{Program} &::= \mathit{Method}^{*} \\
\mathit{Method} &::= f(x_1, \ldots, x_n)\{\mathit{Stmt}\mathbf{;}^{*}\} \\
\mathit{Stmt} &::= \mathit{label}:\ \mathit{Stmt} \mid \mathbf{goto}\,\{t \to \mathit{label}\}^{+} \mid x := t \\
&\quad \mid (x_1, \ldots, x_n) := \mathbf{pull}(p) \mid \mathbf{push}(p, t_1, \ldots, t_n) \\
&\quad \mid x := \mathbf{pull}(ar, t_{idx}) \mid \mathbf{push}(ar, t_{idx}, t) \\
&\quad \mid (x_1, \ldots, x_m) := f(t_1, \ldots, t_n) \mid \mathbf{return}\ (t_1, \ldots, t_m) \mid p := \mathbf{new}\ \mathit{type} \\
&\quad \mid \mathbf{assume}(t) \mid \mathbf{assert}(t) \mid \mathbf{havoc}(x_1, \ldots, x_n)
\end{aligned}
$$

Figure 2: The syntax of our IVL. Here, t is an arithmetic expression; x is a local variable; p is a local variable of reference type; ar is a local variable of array type; *type* is a class type.

3 Optimizing Programs for Verification

We first define a simplified programming language used to represent programs for the purpose of verification. Our Intermediate Verification Language (IVL) is simple, but expressive enough to encode a reasonable subset of Java (without threads or reflection). It serves as the platform for simple static analysis, as well as to encode the results of that analysis to later be leveraged by the solver. We do not go into details of the translation of Java to this language, but instead refer the interested reader to [15], a tool paper about JayHorn.

3.1 Intermediate Verification Language

In our IVL, local variables are either of numeric type (mathematical integers), or of reference type p. Every reference p is a tuple of a heap address id, dynamic type *type*, and the final fields of the object. Every class type *type* is associated with a set $f_1 \ldots f_n$ of fields of either integer or reference type.

We define the syntax of our IVL in Figure 2. Throughout the paper, we use the following conventions: x, y, z refer to local variables; p, r are local variables of reference type, and t, s are side-effect-free expressions over variables and constants. A program in our language consists of a set of methods (functions) with one distinct `main` method. Each method consists of a sequence of (possibly labeled) statements. Labels are symbolic representations of the program counter associated with a particular statement. The first statement in the sequence is the entry point of the method; we make the further simplifying assumption that all methods have a result, and that the last statement in a method is a **return** statement (but the method might contain multiple **return**s).

Control flows from one statement to the next as the program counter increases, or to a specific location if the condition t in a **goto** evaluates to *true*. Assignment statements $x := t$ update a variable x to the value of t. The statement $(x_1, \ldots, x_n) := \mathbf{pull}(p)$ loads the values of all fields $p.f_1, \ldots, p.f_n$ of p into local variables $x_1, \ldots, x_n$. Conversely, $\mathbf{push}(p, t_1, \ldots, t_n)$ updates the fields of p to the values of expressions $t_1, \ldots, t_n$. For arrays, **pull** and **push** includes the index to read or write, respectively. Note that the *length* field of an array is immutable and therefore part of the reference tuple, so it does not need to be accessed via **pull** and **push**. A function call $(y_1, \ldots, y_m) := f(t_1, \ldots, t_n)$ to a method $f(x_1, \ldots, x_n)$ executes the body of the method f until a $\mathbf{return}\ (t_1, \ldots, t_m)$ statement is reached, at which point control returns and the values of the expressions $ret_1, \ldots, ret_m$ are assigned to $y_1, \ldots, y_m$. A new object of class type *type* is allocated with **new** *type*. To be able to distinguish objects on the heap (also from *null*), each allocated object gets a unique id. The statement **assume**(t) blocks the execution if t evaluates to *false* on the current state and has no effect otherwise. The statement **assert**(t) terminates the execution of a program exceptionally if t evaluates to *false* on the current state, and has no effect otherwise. The statement **havoc**$(x_1, \ldots, x_n)$ assigns non-deterministic values to the variables $x_1, \ldots, x_n$.

For class types, we assume a subtyping relationship similar to that in Java. A subtype *type*′ of a type *type* contains all fields $f_1, \ldots, f_n$ of *type*, but can possibly add fields $f_{n+1}, \ldots, f_{n'}$. The dynamic type of a reference is stored in its tuple representation.

In addition, a variable of reference type *type* might actually point to objects of any subtype *type*′, which implies the need for dynamic method dispatch as usual in object-oriented languages. Dynamic dispatching of method invocation may however be replaced with a conditional over the dynamic type of the object, which we assume has been done in the translation to the IVL. To handle object state, we adopt the philosophy that invariants ϕ_{type} only hold for objects that have *exactly* type *type*, i.e., not for objects of subtypes.

A path in a program is a sequence of statements starting from the first statement in the `main` method of a program. A trace is an initial state and the resulting path. We say a trace terminates if its path is finite. A trace terminates erroneously if its path ends with an assertion violation. A trace is infeasible if its path ends with an assumption violation. In this paper, we say that a program is correct if none of its traces terminates erroneously.

3.2 Extensions and Optimizations

A number of simple extensions are possible that significantly increase the precision of the overall space invariants approach, and that can be integrated very organically. We introduce some of the most important techniques in this section.

Optimized Push/Pull Placement As a first optimization of our framework, it is natural to minimize the number of required heap interactions by eliminating redundant **pull** and **push** instructions. Unnecessary heap interaction is detrimental to precision because **push** followed by **pull** from the same object (after translation to space invariants, as outlined in Section 2) will lose information about the precise values of fields. To this end, we apply a simplification step where we use data-flow analysis techniques to remove as much heap interaction as possible.

When a **pull** or **push** is immediately followed by an identical statement, one may be removed. When a **pull** is immediately followed by a **push** of the same local variables and the same object, then the **push** may be removed. Dually, when a **push** is immediately followed by a **pull** of the same object, the **pull** may be removed. In case (some of) the local variables used are different, assignments from the **push**ed values to the variables used in the **pull** must be added. A **pull** can move up in the program (towards the entry), past any statement S for which it can be determined statically that S does not affect the **pull**ed object. A **push** can move down under the same conditions, with the addition that none of the pushed expression may be affected by S.

Note that these optimization rules all depend on data-flow analysis of the IVL program, in particular a points-to analysis. The question whether two references point to the same location on the heap is not decidable in general. The effectiveness of this optimization step depends directly on the precision of the data-flow analysis.

```
// Push site 1
push(p, 0)
goto {b → l₁, ¬b → l₂}

l₁: x := pull(p)
// Push site 2
push(p, x + 1)
goto l₃

l₂: x := pull(p)
// Push site 3
push(p, x + 2)
goto l₃

l₃: x := pull(p)
assert(x > 0)
```

(a) Diamond-shaped code

```
// Push site 1
push(p, 0, 1)
goto {b → l₁, ¬b → l₂}

l₁: (x, lastPush) := pull(p)
assume(lastPush == 1)
// Push site 2
push(p, x + 1, 2)
goto l₃

l₂: (x, lastPush) := pull(p)
assume(lastPush == 1)
// Push site 3
push(p, x + 2, 3)
goto l₃

l₃: (x, lastPush) := pull(p)
assume(lastPush == 2 || lastPush == 3)
assert(x > 0)
```

(b) Code with flow-sensitive invariants

Figure 3: Flow-sensitive space invariants.

Flow-Sensitive Space Invariants A related extension concerns the order in which updates to objects are performed. For any concrete program, the order of statements determines which **push** instructions can influence a **pull**, i.e., where the data read by **pull** can originate from, which can be exploited to make space invariants flow-sensitive.

Fig. 3 illustrates the situation. The program in Fig. 3a is correct, but cannot be verified with naïve translation to space invariants, since the first **push** implies that 0 is a possible value of the field x; this violates the assertion. It is clearly the case that the **pull** at l_3 can only read from push sites 2 or 3, and that the assertion therefore holds in any actual program execution.

To improve precision and handle this situation, we can add a numeric ghost field *lastPush* to every type *type* storing the index of the last push site that updated the object. Standard data-flow analysis can be used to over-estimate the set of **push** statements that might influence a **pull**: for each $l := \mathbf{pull}(p)$ we compute a set of statements $s = \mathbf{push}(q, \ldots)$ such that s occurs on a feasible trace before l, references p, q may alias, and in between l and s there is no $\mathbf{push}(q', \ldots)$ such that q, q' must alias.

We can then exclude influence of other **push** statements by adding an assumption that *lastPush* must be in that set. The instrumented code is shown in Fig. 3b. A Horn solver can now determine a disjunctive invariant for x, where $x > 0$ in case the last push was 2 or 3. As other cases are excluded by the assumption, the assertion can now be proved correct.

Method Inlining As a general preprocessing step that supports the other optimizations, methods that are small and/or are only called few times can be inlined.

Optimization of push/pull placement is applied intra-procedurally. Inlining therefore increases the effectiveness of the optimization by allowing simplification over statements from invoked methods, too.

Furthermore, when a method is invoked from multiple locations, inlining is a means to distinguish between these calls. This increases precision for flow-sensitive space invariants. For instance, if `o.setField(i)` is invoked twice, inlining ensures that there will be two different **push** statements. Data flow analysis will be able to use that information in order to more precisely state which of these can influence a later **pull** from object `o`.

Array Invariants As arrays are difficult to handle for solvers, we avoid using an array theory in our Horn clauses. Instead, we extend space invariants with the array index that is accessed. We thus use $\mathbf{push}(ar, t_{idx}, t)$ to write the value of expression t to array ar at index t_{idx}. Analogously, we read an array location with $x \leftarrow \mathbf{pull}(ar, t_{idx})$. This allows the construction of invariants that are disjunctive with respect to the accessed index. Leveraging the other improvements presented in this section, this enables proofs for many programs with arrays, e.g. our running example.

A multi-dimensional array of dimension $d > 1$ is accessed by first reading/writing the outer array, then treating the value at the given index as an array of dimension $d - 1$. For instance, an array read $x = ar[i][j]$ is translated to $ar_i \leftarrow \mathbf{pull}(ar, i); x \leftarrow \mathbf{pull}(ar_i, j)$.

4 Generating Constrained Horn Clauses

The representation of heap accesses using **push** and **pull** enables us to make the step from precise program execution semantics to summarization of states using space invariants. For this we need a further assumption about the considered programs, namely we rule out programs that can access uninitialized memory:

Definition 1. *We say that a program P* does not access uninitialized memory *if, on every path through P, every $(x_1, \ldots, x_n) := \mathbf{pull}(p)$ is preceded by a $\mathbf{push}(q, t_1, \ldots, t_n)$ where $p{\downarrow}_{id} = q{\downarrow}_{id}$.*

Although it is in general undecidable whether a program can access uninitialized memory, in practice the type system of higher-level languages (in particular of Java) prevents such accesses. For the purpose of verification, the convention does not lead to a loss of generality either, since programs can always start with an initialization phase in which heap data structures are (non-deterministically) created. We can, thus, assume the property from Def. 1.

It is then possible to replace **push**/**pull** pairs with symbolic invariants representing possible states of objects of the accessed type. Suppose a reference tuple has elements $(id, type, f_1, \ldots, f_m)$ and objects of type *type* have additional fields $f_{m+1}, \ldots, f_n$ (remember that only final non-recursive fields go in the tuple). A space invariant is a formula ϕ_{type} over pairwise distinct (fresh) variables $id, x_1, \ldots, x_n$, and will be used to capture the possible states of an object of type *type* at address id. As a convention, for expressions $id', t_1, \ldots, t_n$, we write $\phi_{type}(id', t_1, \ldots, t_n)$ for the result of the substitution $\phi_{type}[id/id', x_1/t_1, \ldots, x_n/t_n]$.

The elimination of **push** and **pull** through the space invariant ϕ_{type} is performed by exhaustive application of the following replacement rules (replace statements above line with statements below line):

$$\frac{\mathbf{push}(p, t_{m+1}, \ldots, t_n)}{\mathbf{assert}(\phi_{p{\downarrow}_{type}}(p{\downarrow}_{id}, p{\downarrow}_{f_1}, \ldots, p{\downarrow}_{f_m}, t_{m+1}, \ldots, t_n))} \tag{1}$$

$$\frac{(x_{m+1}, \ldots, x_n) := \mathbf{pull}(p)}{\mathbf{havoc}(x_{m+1}, \ldots, x_n);\ \mathbf{assume}(\phi_{p{\downarrow}_{type}}(p{\downarrow}_{id}, p{\downarrow}_{f_1}, \ldots, p{\downarrow}_{f_m}, x_{m+1}, \ldots, x_n))} \tag{2}$$

In other words, instead of actually writing data to memory, the resulting program asserts that the data satisfies the stipulated invariant ϕ_{type}; reading data from memory is translated to generating arbitrary values satisfying the invariant. This transformation is *sound:*

Lemma 1. *Let P_{PP} be a program in our IVL, and P_{inv} the result of exhaustively applying* (1) *and* (2)*. If P_{PP} does not access uninitialized memory, then P_{PP} is correct if P_{inv} is correct (but in general not vice versa).*

After replacing memory access via **pull** and **push** by space invariants, our IVL only contains local variables and translation into logic becomes simple. To prove the safety of all assertions in a program, we encode the program into a set of constrained Horn clauses (CHC).

A constrained *Horn clause* is a formula $C \wedge B_1 \wedge \cdots \wedge B_n \to H$ where C is a constraint over variables and interpreted predicates (e.g. $>$, or $+$). Each B_i is an application of a relational symbol $p(t_1, \ldots, t_n)$ to terms t_i in first-order logic. We refer to $C \wedge B_1 \wedge \cdots \wedge B_n$ as the *body* of a Horn clause. The *head* of a Horn clause H is, similar to B_i, an application $p(t_1, \ldots, t_n)$, or *false*. A set of CHCs is called *solvable* if there is an instantiation of the used relational symbols for which all Horn clauses are valid. As shown in [3], the assertion checking problem can be reduced to solvability of Horn clauses.

Suppose a method $f(x_1, \ldots, x_n)$ with n arguments and k results. To represent the contract of f, we follow the standard encoding (see, e.g., [12]) and assume two predicates pre_f and $post_f$, where pre_f has arity n and $post_f$ has arity $n + k$; those predicates will represent the precondition and postcondition of f.

We further assume one predicate pc_i per program location $pc = i$. The arity $m = n + l$ of this symbol is determined by the number n of arguments of f, and the number l of program variables that can be alive at this program point (live variables can be computed statically). In addition to that, we create one predicate ϕ_{type} for each *type* used in the program. The arity $k+1$ of ϕ_{type} is determined by the number k of fields in *type* (one additional argument is needed for the object reference). These predicates are shared between all methods.

For the method entry point, we introduce one Horn clause connecting the precondition with the method entry $pc = 1$:

$$pre_f(x_1, \ldots, x_n) \;\to\; pc_1(x_1, \ldots, x_n, \ldots)$$

For each statement s at program counter $pc = i$ whose execution transitions to $pc = j$, we can then create Horn clauses of the general shape:

$$pc_i(x_1, \ldots, x_n, \ldots) \wedge \psi \;\to\; pc_j(x_1, \ldots, x_n, \ldots)$$

The method arguments are always carried through in this clause (since postconditions can refer to the method arguments), while further predicate arguments, and the guard ψ, depend on the nature of the statement s. Intuitively, the Horn clause expresses that, if we are at program counter i and the invariant pc_i and the condition ψ hold, then we can transition to program counter j where the invariant pc_j has to hold. Additional clauses are introduced for method calls, **return**, and **assert** statements.

The translation from the individual statements to clauses is mostly straightforward, we only show the most interesting cases.

A statement **assert**(ϕ) at $pc = i$ is translated to two Horn clauses:

$$\begin{aligned} pc_i(x_1, \ldots, x_n, y_1, \ldots, y_l) \wedge \neg\phi \;&\to\; \mathit{false} \\ pc_i(x_1, \ldots, x_n, y_1, \ldots, y_l) \;&\to\; pc_{i+1}(x_1, \ldots, x_n, \ldots) \end{aligned} \tag{3}$$

where we assume that ϕ is formulated over the live variables $y_1, \ldots, y_l$.

As a special case, for a statement **assert**$(\phi(p, t_1 \ldots t_n))$ introduced by (1), clause (3) is turned into an implication

$$pc_i(x_1, \ldots, x_n, y_1, \ldots, y_l) \;\rightarrow\; \phi_{type}(p, t_1 \ldots t_n)$$

A call $(y_1, \ldots, y_{k'}) := g(t_1, \ldots, t_{n'})$ to method g is represented by two clauses, one asserting the precondition of g, and one applying the postcondition:

$$\begin{array}{rcl} pc_i(x_1, \ldots, x_n, y_1, \ldots, y_l) & \rightarrow & pre_g(t_1, \ldots, t_{n'}) \\ \begin{array}{l} pc_i(x_1, \ldots, x_n, y_1, \ldots, y_l) \wedge \\ post_g(t_1, \ldots, t_{n'}, r_1, \ldots, r_{k'}) \end{array} & \rightarrow & pc_{i+1}(x_1, \ldots, x_k, \ldots, r_1, \ldots, r_{k'}, \ldots) \end{array}$$

Expressions $t_1, \ldots, t_{n'}$ are again formulated over live variables $y_1, \ldots, y_l$, while the result variables $r_1, \ldots, r_{k'}$ are used in the second clause to update relevant live variables in the post-state.

A return statement **return** $(t_1, \ldots, t_k)$ at $pc = i$ is dually represented as a clause asserting the postcondition:

$$pc_i(x_1, \ldots, x_n, y_1, \ldots, y_l) \;\rightarrow\; post_f(x_1, \ldots, x_n, t_1, \ldots, t_k)$$

Finally, we add one more Horn clause: $\mathit{true} \rightarrow pre_{\mathrm{main}}$ to assert that the precondition of `main` has to be *true* (i.e., no input can violate an assertion). In practice, we have to add a prelude to the Horn clauses representing `main`. The representation discussed above assumes that the execution of the program starts with a completely empty heap but, for example, the `main` method of a Java program takes an array of strings as input that exists prior to the execution of `main`. Hence, we need to add a short prelude that allocates this array and fills it with non-deterministic values.

5 Implementation and Experimental Evaluation

The encoding of Java into our IVL and from there to Horn clauses is implemented in JayHorn[6]. JayHorn takes Java bytecode as input and tries to prove that no exception of a pre-defined set can leave the `main` method. This includes user-defined exceptions, Java assertions (which are represented as exceptions in bytecode) and implicit exceptions that are thrown by null-pointer dereferences, array bounds violations, and illegal casts. More details on the implementation are given in [15].

In its default configuration, JayHorn checks for null-pointer, array bounds, and cast exceptions, as well as any non-runtime exception. Library calls are assumed to not throw exceptions. That is, in its default configuration, JayHorn would not check if `parseInt` in our example from Section 2 throws an exception. This is a convenient assumption because we do not have to model libraries per se, but can simply havoc all local variables a library can may touch. If needed, JayHorn can be configured to be either pedantic and assume that library calls always may throw explicitly declared exceptions, or to provide summaries in the form of Horn clauses.

JayHorn supports two solver backends, Eldarica and Spacer. In its default configuration, JayHorn uses Eldarica and can be run from a single fat Jar. The implementation is modular to ensure that it can be extended to support other Horn solver backends with reasonable effort.

Soundness. JayHorn is sound w.r.t. programs written in the intermediate verification language and soundy [20] for single-threaded Java. All benchmarks used below only use Java features for

[6] http://jayhorn.github.io/jayhorn/

which JayHorn is sound, but some language features are not supported (see website for details). Notably, reflection is not supported and the current implementation has limited support for static initializers.
Completeness. JayHorn is not complete. We trade completeness for efficiency (i.e. the ability to verify more programs faster). In addition to inherently undecidable problems, other sources of incompleteness are numeric types other than integer, restricted expressiveness of space invariants, coarse abstraction of library calls, etc.

Evaluation. In the following we evaluate how JayHorn performs on different verification problems and how the two JayHorn backend solvers Spacer and Eldarica perform. For the evaluation, we use three sets of benchmarks[7].

The first set is our own benchmark set. We have compiled a set of 82 benchmark problems, available in the JayHorn repository. 20 of the benchmarks are synthetic problems, such as calculation of Fibonacci numbers or the Ackermann function; the rest are algorithms and data-structures implementations (e.g., sorted binary trees) that require non-trivial invariants.

The second benchmark set is the MinePump set provided by CPAChecker [2]. This is a Java version of a well known benchmark set for C program verification. It contains 64 different versions of a MinePump controller and its environment together with safety properties that need to be verified.

We also compare JayHorn with the CPAChecker [2] tool, which combines model-checking with other program analysis techniques. CPAChecker currently only handles a smaller subset of Java than JayHorn, so the results cannot be generalized.

The third benchmark set is provided by CBMC. It contains 42 Java programs with assertions. The benchmarks mostly test support of language features, such as sub-typing, casting between numeric types, etc. For our evaluation, we did not compare with the Java version of CBMC since the version of this tool that we evaluated was limited to only handle single class files without dependencies and thus could not be run on most of our benchmarks.

Discussion. Table 1 shows the result of running JayHorn and CPAChecker on the three benchmark sets. As stated above, not all benchmarks are suitable for all tools and hence we will not draw conclusions about the capabilities of the other tools from benchmarks on which they do not perform well.

The first observation is that JayHorn can solve almost 90% of the benchmarks. It solves these within one minute (increasing the time-out to 60 minutes does not help solve the remaining problems). This supports our claim that our encoding using space invariants is an effective and lightweight way to verify realistic problems.

Our incompleteness caused imprecise results for less than 8% of the benchmarks. Most of the imprecise results are reported on MinePump. We manually inspected these cases and came to the following conclusion: the MinePump benchmarks implement a state machine whose state is encoded in an `Enum` that is set in different method calls. When computing a space invariant for the field that holds the state of the pump, JayHorn over-approximates the possible states the pump can be in. One way to address this is to fine-tune the inlining of methods in JayHorn to this example. Without manual fine-tuning, JayHorn is slower and solves fewer of these problems than CPAChecker.

On the other examples, JayHorn gives many more correct answers than CPAChecker. This is in part due to Java language features that are currently not supported by the CPAChecker.

[7]Benchmarks used for the experiments can be found in `https://github.com/jayhorn/benchmarks/tree/master/lpar17-benchmarks`

Table 1: Experimental results. We show, for each set of benchmarks and for both tools, the number of cases where the tool's result is correct or incorrect, distinguishing cases where the expected answer for the program is safe or unsafe. Note that while incorrectly answering that a program is unsafe is a matter of precision (the program cannot be proved correct), incorrectly answering that a program is safe means that the tool is not sound. The "TO" column lists the benchmarks for which the tool timed out after 60 seconds. The "N/A" lists benchmarks that are not applicable to a tool, since they use unsupported language features.

		Correct		**Incorrect**		**TO**	**N/A**
		Safe	Unsafe	Imprecise	Unsound		
JayHorn benchmarks (41 safe, 41 unsafe)	JayHorn (Eldarica)	33	41	3	0	5	0
	JayHorn (Spacer)	37	41	3	0	1	0
	CPAChecker	5	5	0	0	0	72
MinePump (43 safe, 21 unsafe)	JayHorn (Eldarica)	32	21	11	0	0	0
	JayHorn (Spacer)	32	19	8	0	5	0
	CPAChecker	43	21	0	0	0	0
CBMC benchmarks (35 safe, 7 unsafe)	JayHorn (Eldarica)	29	7	3	0	0	3
	JayHorn (Spacer)	29	7	3	0	0	3
	CPAChecker	6	0	0	0	0	36
Total (119 safe, 76 unsafe)	JayHorn (Eldarica)	94	69	17	0	5	3
	JayHorn (Spacer)	98	67	14	0	6	3
	CPAChecker	54	26	0	0	0	108

For three of the CBMC benchmarks, JayHorn could not parse the input. We assume that these benchmarks are malformed since the Soot bytecode parser rejected them. For three other benchmarks, JayHorn was incomplete. These benchmarks contain double and float types, which are currently treated as opaque objects in JayHorn.

Overall, the Spacer backend performed slightly better than Eldarica. This is largely due to preprocessing and simplification tricks that Spacer uses to simplify the problem. For example, if we modify our program from Section 2 by setting the `size` variable to the length of `args`, Eldarica is not able to solve the problem anymore because it can't use the length field of `args` in the space invariant for Node, but Spacer is able to solve it by introducing a ghost symbol.

In summary, we can say that space invariants are a practical approach to verify programs that allocate an unbounded amount of heap. The approach is able to handle many practical problems, and its incompleteness does not significantly limit the number of problems that can be solved.

6 Related Work

Numerous approaches have been presented that generate quantified inductive invariants. Our approach has some similarities with the work of Bjørner et. al. [4], in which a Horn solver is guided by constraining the form of the proof, enabling it to find suitable (quantified) invariants even in challenging cases. A related technique for approximation of array behavior was recently presented in [23]. A number of abstract interpretation methods have domains that represent universally quantified facts [9, 13]. In our approach, like in [4] we aim to avoid the explicit construction of abstract post operators, widening and refinement procedures needed in these approaches. The work in [18] synthesizes a class of universally quantified linear invariants of array programs. Our technique aims to synthesize a general class of invariants.

Other approaches, such as [21] and [25] provide semi-automated ways to analyze program with unbounded data. Our approach does not require user-provided information.

At a very high level, we are inspired by techniques based on dependent refinement and liquid types [28]. Variations of refinement types and their applications have been studied in many settings (e.g. [1, 11, 16, 31]). Liquid types [28] allow automatic inference of dependent types precise enough to prove a variety of properties. Our approach is formulated in terms of invariants (in contrast to types), and mainly focuses on derivation of heap invariants for object-oriented programs.

The use of push and pull in our approach is similar to *unfold/fold* techniques used for program transformation [6] on purely functional programs. A similar technique called *unroll/roll* was later used in alias types [30] to manually witness the isomorphism between a recursive type and its unfolding. The Viper [24] tool uses the constructs inhale and exhale which are similar in spirit to our pull and push operations but their approach targets permission logic which is not immediately comparable.

Separation logic [14, 27] is widely used to reason about heap structure. Bi-abduction [8] based tools such as jStar [10] or Infer [7] have delivered impressive results. Our approach is orthogonal in the sense that we reason about heap data rather than heap structure.

7 Conclusion

We have introduced a new methodology to verify object-oriented programs with the help of space invariants. We have demonstrated an encoding of Java programs into Horn clauses that allows a Horn solver to automatically infer space invariants and verify non-trivial examples.

We have demonstrated an implementation that is capable of solving problems fully automatically that cannot be solved easily with other encodings. Our experiments show that the encoding is sufficiently general to deliver good results with the Horn solvers Eldarica and Spacer.

In the future we plan to introduce invariants that can speak about multiple objects to help the Horn solver to find more advanced properties, such as sortedness. Adding additional predicates can also help to restore completeness which is an essential part of our future work.

We believe that space invariants give rise to an elegant, lightweight and scalable method of encoding heap access in Java-like programs. They are easy to implement, have a high potential for automation, and can verify interesting properties that are expensive to show with more heavyweight approaches.

The basic approach of space invariants through **pull** and **push** instructions is introduced, along with a series of refinements. The presented technique has been implemented in the tool JayHorn, the effectiveness of which is demonstrated on a large set of benchmarks.

To summarize: we have demonstrated a new encoding of Java-like languages into Horn clauses that allows the Horn solver to infer space invariants that summarize heap regions and make it possible to find proofs for non-trivial programs. The approach is implemented in JayHorn and available for download.

References

[1] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008*, pages 17–32, 2008.

[2] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 184–190, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT Workshop at IJCAR*, 2012.

[4] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *Static Analysis - 20th International Symposium, SAS 2013*, pages 105–125, 2013.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[7] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[8] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN Notices*, volume 44, pages 289–300. ACM, 2009.

[9] Patrick Cousot. Verification by abstract interpretation, soundness and abstract induction. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 1–4, 2015.

[10] Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008*, pages 213–226, 2008.

[11] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, 1991.

[12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416, 2012.

[13] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 235–246, 2008.

[14] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[15] Temesghen Kahsai, Phillip Rummer, Huascar Sanchez, and Martin Schaf. JayHorn: A framework for verifying java programs. In *Computer Aided Verification (CAV'16)*, 2016.

[16] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2), 2010.

[17] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 17–34, 2014.

[18] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. Smt-based array invariant generation. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference,*

VMCAI 2013, pages 169–188, 2013.

[19] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Proceedings of the 2005 International Conference on Formal Methods*, FM'05, pages 26–42, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[21] Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Static Analysis - 17th International Symposium, SAS 2010*, pages 71–99, 2010.

[22] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1988.

[23] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 361–382. Springer, 2016.

[24] 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, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[25] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007*, pages 251–266, 2007.

[26] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper - complete heap verification with mixed specifications. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2014.

[27] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.

[28] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[29] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 347–363, Berlin, Heidelberg, 2013. Springer-Verlag.

[30] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *Types in Compilation, Third International Workshop, TIC 2000*, pages 177–206, 2000.

[31] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, 1999.