Reasoning in the Theory of Heap: Satisfiability and Interpolation

Zafer Esen and Philipp Rümmer

Uppsala University, Sweden

Abstract. In recent work, we have proposed an SMT-LIB theory of heap tailored to Horn-clause verification. The theory makes it possible to factor out the treatment of heap from verification systems, and lift approaches to handle heap-allocated data-structures in verification to a language-independent level. This paper gives an overview of the theory, and presents ongoing research on decision and interpolation procedures.

1 Introduction

Tools for formal program verification are often engineered making use of various existing libraries and frameworks; for instance, compiler front-ends, constraint and SMT solvers, and more recently solvers for Constrained Horn Clauses (CHCs). This way, the effort required to construct verification systems can be reduced significantly, a wider range of languages or applications can be covered, and the quality and performance of the resulting tool is improved. In this paper, we consider the use of Constrained Horn Clauses, which represent an intermediate verification language tailored to the analysis of safety properties, and can be solved by CHC solvers such as Spacer [20] or Eldarica [16]; for an overview see [3, 28]. These solvers in turn utilise theorem provers or SMT solvers such as Z3 [23] or Princess [27] to reason about the constraints in CHCs.

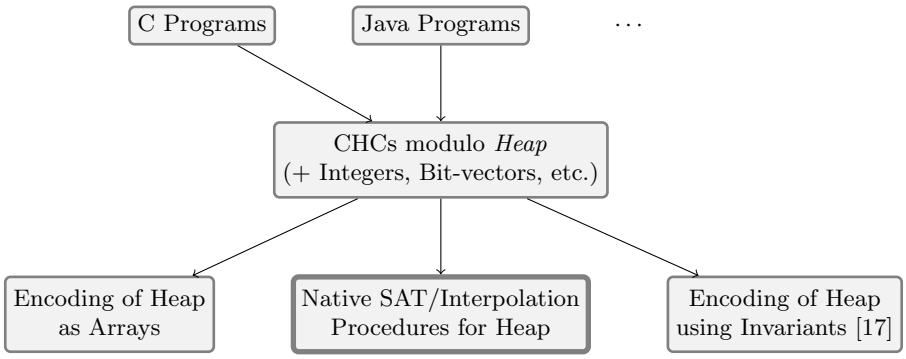

Fig. 1. Program verification using the theory of heap.

A challenging feature of languages, in this context, is the handling of heap-allocated data-structures: such data-structures are today either represented explicitly using the theory of arrays (e.g., [19, 11]), or are transformed away with the help of invariants or refinement types (e.g., [26, 4, 22, 17]). In [12], we motivate the alternative approach of having heap as a native theory supported by solvers, which turns CHCs into a standardised interchange format for programs with heap data-structures. Figure 1 shows the resulting verification flow: verification tools would take programs, for instance in C or Java, as input, and encode them in a uniform way as CHCs modulo the theory of heap. The encoding keeps heap operations like read, write, or allocation essentially intact, and it is up to CHC and SMT solvers to process those operations further. CHC solvers could, e.g., choose to encode heap into arrays, or apply invariant-based encoding.

In this paper, we present first steps of the development of native decision and interpolation procedures for the theory of heap, covering two main reasoning tasks needed to implement CHC solvers [29, 3]. The described procedures are intended as a starting point and are currently largely unoptimised. We expect that many optimisations from array solvers (e.g., [30, 9, 14, 7]) can be adapted.

1.1 Encoding Programs using the Theory of Heap

In Listing 1.1 a C program is given in order to show the intuition behind the encoding and provide an overview of the theory. The program has a single function `insertNode` that allocates and initialises a list `Node` (as defined in line 1), and appends it to the passed list pointed to by `p`.

One way to encode this program is using CHCs, and to consider the heap as a single shared mutable data-structure. A theory of heap provides *Heap* and *Addr* sorts, so *Heap* and *Addr* terms can be used in the CHCs just as any other term. A diagram illustrating the effect of the CHCs is given in Figure 2. As an example, the statement at line 4 of Listing 1.1 can be encoded using the topmost constraint on the right-hand side of the diagram, which allocates a `Node` with uninitialised fields. A CHC can then be constructed using the invariants I_1 and I_2 that encode program state and the constraint C_1 that encodes the transition as $I_1(...) \wedge C_1 \rightarrow I_2(...)$, where the dots "..." represent the program variables in scope along with the heap term.

A complete SMT-LIB encoding of the program from Listing 1.1 is given in Listing 1.2. Lines 1-9 show the heap declaration in SMT-LIB format, where:

- *Heap* and *Addr* are the names of the declared heap and address sorts,
- *Object* is the name of the *selected* object sort,
- `Node` and `Object` are the declared data-types,
- `O_Empty` is the default *Object* term that is returned on invalid reads.

The object sort is said to be *selected*, because it could also be declared outside the heap declaration and only specified here. This is not possible in this example as one of the declared data-types (`Node`) has a field that points to an address on the heap (i.e., *Addr*), and the address sort only becomes available with the heap declaration.

Listing 1.1. A C function that adds a new node to the head of a linked list.

```
1  struct Node { int data; struct Node* next; };
2
3  void insertNode (int d, struct Node* p) {
4    struct Node* n = malloc(sizeof(struct Node));
5    n->data = d;
6    n->next = p->next;
7    p->next = n;
8  }
```

Listing 1.2. Complete encoding of the program from Listing 1.1. The heap declaration is given in SMT-LIB notation, while the clauses and the assertions are given in Prolog notation. ":-" corresponds to a left implication arrow (i.e., "←"), and it is assumed that all variables occurring in the clauses are universally quantified with the correct sort (i.e., $\forall h : Heap.\ \forall p : Addr.\ \ldots$).

```
1  (declare-heap
2   Heap                                ; declared Heap sort
3   Addr                                ; declared Address sort
4   Object                              ; chosen Object sort
5   O_Empty                             ; the default Object
6   ((Node 0) (Object 0))               ; ADTs
7   (((Node   (data Int) (next Addr)))  ; Class constructors
8    ((O_Node (getNode Node))           ; Object sort constructors
9     (O_Empty))))
10                                      ; invariant declarations
11 (declare-fun I1 (Heap Int Addr)           Bool) ; <h,d,p>
12 (declare-fun I2 (Heap Int Addr Addr)      Bool) ; <h,d,p,n>
13 (declare-fun I3 (Heap Int Addr Addr)      Bool) ; <h,d,p,n>
14 (declare-fun I4 (Heap Int Addr Addr Addr) Bool) ; <h,d,p,n,t>
15 (declare-fun I5 (Heap Int Addr Addr)      Bool) ; <h,d,p,n>
16 (declare-fun I6 (Heap Int Addr)           Bool) ; <h,d,p>
17
18 ; Clauses (given in Prolog notation for readability)
19 I1(h,d,p)     :- h = emptyHeap.
20
21 I2(h',d,p,n)  :- I1(h,d,p),
22                   h' = newHeap(alloc(h, O_Node(_nonDet))),
23                   n = newAddr(alloc(h, O_Node(_nonDet))).
24
25 I3(h',d,p,n)  :- I2(h,d,p,n),
26                   h'= write(h,n,
27                     O_Node(Node(d,next(getNode(read(h,n)))))).
28
29 I4(h,d,p,n,t) :- I3(h,d,p,n), t = next(getNode(read(h,p))).
30
31 I5(h',d,p,n)  :- I4(h,d,p,n,t),
32                   h'= write(h,n,
33                     O_Node(Node(data(getNode(read(h,n))),t))).
34
35 I6(h',d, p)   :- I5(h,d,p,n),
36                   h'= write(h,p,
37                     O_Node(Node(data(getNode(read(h,p))),n))).
38
39 ; Assertions
40 false :- I2(h,d,p,n),   !is_O_Node(read(h,n)).
41 false :- I3(h,d,p,n),   !is_O_Node(read(h,p)). ; <- will fail
42 false :- I4(h,d,p,n,t), !is_O_Node(read(h,n)).
43 false :- I5(h,d,p,n),   !is_O_Node(read(h,p)). ; <- will fail
```

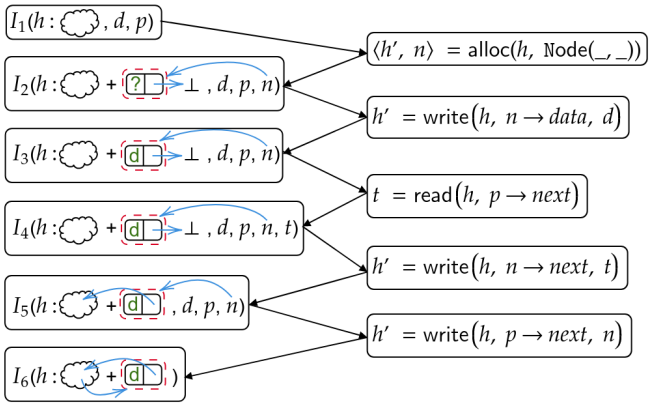

Fig. 2. The boxes on the left-hand side correspond to invariants encoding the state of a program at a certain point. E.g., I_1 is the entry point of `insertNode`, where the only relevant variables in scope are the arguments of the function, and the heap. The constraints on the right-hand side correspond to executing program statements. Black arrows visualise the execution of the program. h is the *Heap* term representing the heap. A h' term in the constraints is equal to the h term of the next invariant. An underscore represents a term that can have any value (of the correct sort within the context). t is a fresh *Addr* variable. A blue arrow visualises where a pointer is pointing to, and a red-dashed line symbolises an *Object*.

Specifying a single object sort makes it possible to have a unified sort on the heap, and the flexibility of algebraic data-types (ADTs) simplifies encoding many programming language types.

Lines 11-16 declare the invariants which are used in the CHCs. The rest of the encoding shows the CHCs in Prolog notation as explained in Figure 2.

The assertions at lines 36-39 check the validity and type safety of heap accesses. These make use of testers that come with algebraic data-types. It can be seen that in this case the assertions at lines 37 and 39 will fail, because the first system transition at line 19 starts with the emptyHeap, which has no valid addresses that `p` can point to.

2 Preliminaries

2.1 The Theory of Heap

Signature Functions and predicates of the theory are given in Table 1.

nullAddr returns an *Addr* which is unallocated (or invalid) in all heaps.
emptyHeap returns the *Heap* that is unallocated everywhere.
allocate takes a *Heap* and an *Object*, and returns an *AllocationResultHeap*. *AllocationResultHeap* is an ADT representing the pair $\langle Heap, Addr \rangle$. The returned *Heap* contains the passed *Object* at *Addr*.

nullAddr : ()	$\rightarrow Addr$
emptyHeap : ()	$\rightarrow Heap$
allocate : $Heap \times Object$	$\rightarrow Heap \times Addr$ ($AllocationResultHeap$)
valid : $Heap \times Addr$	$\rightarrow Boolean$
read : $Heap \times Addr$	$\rightarrow Object$
write : $Heap \times Addr \times Object$	$\rightarrow Heap$

Table 1. Functions and predicates of the theory of heap.

$I(Heap) = I(Object)^*$
$I(Addr) = \mathbb{N}$

$I(\mathsf{nullAddr})$	$= 0$
$I(\mathsf{emptyHeap})$	$= \epsilon$
$I(\mathsf{read})(h, a)$	$= \begin{cases} h[a-1] & \text{if } 0 < a \leq \lvert h\rvert, \\ defObj & \text{otherwise.} \end{cases}$
$I(\mathsf{write})(h, a, o)$	$= \begin{cases} h[a-1 \mapsto o] & \text{if } 0 < a \leq \lvert h\rvert, \\ h & \text{otherwise.} \end{cases}$
$I(\mathsf{allocate})(h, o)$	$= \langle h +\!\!+ [o], \lvert h\rvert + 1\rangle$
$I(\mathsf{valid})(h, a)$	$= 0 < a \leq \lvert h\rvert$

Table 2. Interpretation of sorts, functions, and predicates in the theory of heap. The symbol $+\!\!+$ denotes concatenation of two sequences.

valid is the predicate checking if the passed *Addr* is allocated in the passed *Heap*.
If it is allocated then we say that an access is *valid*; it is *invalid* otherwise.

read returns the *Object* at the passed *Addr* of the passed *Heap* on a valid access.
If the access is invalid, then the specified default *Object* is returned instead (line 5 in Listing 1.2).

write, if the access is valid, returns a heap where the passed *Addr* of the passed *Heap* is updated with the passed *Object*, with all other locations unchanged.
If the access is invalid, the passed heap is returned without any changes.

Semantics A many-sorted signature can be defined as the triple $L = \langle S, \Sigma_f, \Sigma_p \rangle$ where S is a set of sorts, Σ_f is a set of function symbols and Σ_p is a set of relation symbols. A structure is a pair $\langle D, I \rangle$ where D is the domain (consisting of disjoint subsets for each sort in S), and I is an interpretation function that associates each $f \in \Sigma_f$ and $p \in \Sigma_p$ to some n-ary function or relation. Arguments of both f and p and the values of f are specified using the sorts in S.

The heap is interpreted as an ordered sequence of zero or more heap objects. The sort *Object* can in principle be any selected sort, but will in most cases be an ADT, and be interpreted as the set of constructor terms of the ADT. Addresses (*Addr*) are interpreted as natural numbers. $h[k]$ denotes the $(k+1)$–th heap object, where $k \in \mathbb{N}$. Formal definitions are given in Table 2.

2.2 An Interpolating Sequent Calculus for First-Order Logic modulo Integers

We formulate our decision procedure for heap formulas on top of a simple logic of Presburger arithmetic constraints combined with uninterpreted predicates, introduced in [27] and extended in [5, 6] to support Craig interpolation. Since the logic does not support functions, the heap operators (and also ADTs) have to be encoded using relations, with explicit rules for functional consistency; this setting closely models the situation in SMT solvers, where functions are handled by a separate theory implementing congruence closure.

Let x range over an infinite set X of variables, p over a set P of uninterpreted predicates with fixed arity, and α over the set $\mathbb{Z}$ of integers. The syntax of terms and formulae is defined by the following grammar:

$$\phi ::= t = 0 \,\big|\, t \leq 0 \,\big|\, p(t, \ldots, t) \,\big|\, \phi \wedge \phi \,\big|\, \phi \vee \phi \,\big|\, \neg\phi \,\big|\, \forall x.\phi \,\big|\, \exists x.\phi$$
$$t ::= \alpha \,\big|\, x \,\big|\, \alpha t + \cdots + \alpha t$$

The symbol t denotes terms of linear arithmetic. Substitution of a term t for a variable x in ϕ is denoted by $[x/t]\phi$; we assume that variable capture is avoided by renaming bound variables as necessary. For simplicity, we sometimes write $s = t$ as a shorthand of $s - t = 0$, and the inequality $s \leq t$ for $s - t \leq 0$. The abbreviation *true* (*false*) stands for the equality $0 = 0$ ($1 = 0$), and the formula $\phi \rightarrow \psi$ abbreviates $\neg\phi \vee \psi$. Semantic notions such as structures, models, satisfiability, and validity are defined as is common (e.g., [13]), but we assume that evaluation always happens over the universe $\mathbb{Z}$ of integers.

A Sequent Calculus for the Base Logic For checking whether a formula in the base logic is satisfiable or valid, we work with a simplified version of the calculus presented in [27], a part of which is shown in Table 3. If Γ, Δ are sets of formulae, then $\Gamma \vdash \Delta$ is a *sequent*. A sequent is *valid* if the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. Proofs are trees growing upward, in which each node is labelled with a sequent, and each non-leaf node is related to the node(s) directly above it through an application of a calculus rule. A proof is *closed* if it is finite and all leaves are justified by an instance of a rule without premises. Soundness of the calculus implies that the root of a closed proof is a valid sequent.

In addition to propositional and quantifier rules in Table 3, the calculus in [27] also includes rules for equations and inequalities in Presburger arithmetic; the details of those rules are not relevant for this paper.

Craig Interpolation in the Base Logic Given formulas A and B such that $A \wedge B$ is unsatisfiable, Craig interpolation can determine a formula I such that the implications $A \Rightarrow I$ and $B \Rightarrow \neg I$ hold, and non-logical symbols in I occur in both A and B [10]. An interpolating version of our sequent calculus has been presented in [5, 6], and is summarised in Table 4. To keep track of the partitions A, B, the calculus operates on labelled formulas $\lfloor \phi \rfloor_L$ (with L for "left") to indicate that

$$\frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \vee \psi \vdash \Delta} \vee\text{-LEFT} \qquad \frac{\Gamma, \phi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \wedge\text{-RIGHT}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \wedge\text{-LEFT} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta} \vee\text{-RIGHT}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta} \neg\text{-LEFT} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta} \neg\text{-RIGHT}$$

$$\frac{*}{\Gamma, \phi \vdash \phi, \Delta} \text{CLOSE}$$

$$\frac{\Gamma, [x/t]\phi, \forall x.\phi \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta} \forall\text{-LEFT} \qquad \frac{\Gamma, [x/x']\phi \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta} \exists\text{-LEFT}$$

$$\frac{\Gamma \vdash [x/t]\phi, \exists x.\phi, \Delta}{\Gamma \vdash \exists x.\phi, \Delta} \exists\text{-RIGHT} \qquad \frac{\Gamma \vdash [x/x']\phi, \Delta}{\Gamma \vdash \forall x.\phi, \Delta} \forall\text{-RIGHT}$$

Table 3. A selection of the basic calculus rules for propositional and first-order logic. In the rules $\exists$-LEFT and $\forall$-RIGHT, x' is a variable that does not occur in the conclusion.

ϕ is derived from A, and similarly formulas $\lfloor\phi\rfloor_R$ for ϕ derived from B. If Γ, Δ are finite sets of L/R-labelled formulas, and I is an unlabelled formula, then $\Gamma \vdash \Delta \blacktriangleright I$ is an *interpolating sequent*.

Semantics of interpolating sequents is defined using the projections $\Gamma_L =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_L \in \Gamma\}$ and $\Gamma_R =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_R \in \Gamma\}$, which extract the L/R-parts of a set Γ of labelled formulae. A sequent $\Gamma \vdash \Delta \blacktriangleright I$ is *valid* if 1. the sequent $\Gamma_L \vdash I, \Delta_L$ is valid, 2. the sequent $\Gamma_R, I \vdash \Delta_R$ is valid, and 3. the variables and uninterpreted predicates/functions in I occur in both $\Gamma_L \cup \Delta_L$ and $\Gamma_R \cup \Delta_R$. As a special case, note that the sequent $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$ is valid iff I is an interpolant of $A \wedge B$. Soundness of the calculus guarantees that the root of a closed interpolating proof is a valid interpolating sequent.

To solve an interpolation problem $A \wedge B$, a prover typically first constructs a proof of $A, B \vdash \emptyset$ using the ordinary calculus from Table 3. Once a closed proof has been found, it can be lifted to an interpolating proof: this is done by replacing the root formulas A, B with $\lfloor A\rfloor_L, \lfloor B\rfloor_R$, respectively, and recursively assigning labels to all other formulas as defined by the rules from Table 4. Then, starting from the leaves, intermediate interpolants are computed and propagated back to the root, leading to an interpolating sequent $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \emptyset \blacktriangleright I$.

2.3 Reduction for the Theory of Algebraic Data-Types

The heap theory uses ADTs to represent the objects stored on a heap, which means that a decision procedure for heap formulas also has to handle ADTs. For this purpose, in principle any existing algorithm for ADT formulas can be used, e.g., [18, 2, 31, 25]. In this paper, we make use of the reduction approach for ADT formulas defined in [15], which translates a quantifier-free ADT formula to an equisatisfiable formula in the combined theory of equality and uninterpreted

$$\frac{\Gamma, \lfloor\phi\rfloor_L \vdash \Delta \blacktriangleright I \qquad \Gamma, \lfloor\psi\rfloor_L \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_L \vdash \Delta \blacktriangleright I \vee J} \vee\text{-LEFT}_L$$

$$\frac{\Gamma, \lfloor\phi\rfloor_R \vdash \Delta \blacktriangleright I \qquad \Gamma, \lfloor\psi\rfloor_R \vdash \Delta \blacktriangleright J}{\Gamma, \lfloor\phi \vee \psi\rfloor_R \vdash \Delta \blacktriangleright I \wedge J} \vee\text{-LEFT}_R$$

$$\frac{\Gamma, \lfloor\phi\rfloor_D, \lfloor\psi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\phi \wedge \psi\rfloor_D \vdash \Delta \blacktriangleright I} \wedge\text{-LEFT}_D \qquad \frac{\Gamma \vdash \lfloor\phi\rfloor_D, \Delta \blacktriangleright I}{\Gamma, \lfloor\neg\phi\rfloor_D \vdash \Delta \blacktriangleright I} \neg\text{-LEFT}_D$$

$$\frac{*}{\Gamma, \lfloor\phi\rfloor_L \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright \phi} \text{CLOSE}_{LR} \qquad \frac{*}{\Gamma, \lfloor\phi\rfloor_R \vdash \lfloor\phi\rfloor_R, \Delta \blacktriangleright \mathit{true}} \text{CLOSE}_{RR}$$

$$\frac{\Gamma, \lfloor[x/t]\phi\rfloor_R, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\forall x.\phi\rfloor_R \vdash \Delta \blacktriangleright \exists_{Lt}\, I} \forall\text{-LEFT}_R \qquad \frac{\Gamma, \lfloor[x/x']\phi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\exists x.\phi\rfloor_D \vdash \Delta \blacktriangleright I} \exists\text{-LEFT}_D$$

Table 4. A selection of interpolating rules for propositional and first-order logic. Parameter D stands for either L or R. The quantifier $\exists_{Lt}$ denotes existential quantification over all free variables occurring in t but not in $\Gamma_R \cup \Delta_R$. In $\exists$-LEFT$_D$ and $\forall$-RIGHT$_D$, x' is a fresh variable that does not occur in the conclusion.

functions (EUF) and linear integers (LIA). An EUF+LIA formula can be translated further to a formula in the base logic from the previous section, and this way also Craig interpolants can be computed for ADT formulas.

The details of [15] are not important for the present paper, and we only assume that a function *adtReduction* is available that maps quantifier-free ADT formulas to equisatisfiable formulas in the base language.

3 A Decision Procedure for the Theory of Heap

We now define our calculus and decision procedure for quantifier-free heap formulas. Similarly to the approach chosen in [15, 1], the procedure consists of two components: a set of rewriting rules for translating heap formulas to a core language (Section 3.2, Table 5), and a set of sequent calculus rules for handling the core language (Section 3.3, Table 6).

3.1 The Core Language for Heap Formulas

Our core language for heap constraints is defined on top of first-order logic modulo integers, as introduced in Section 2.2. Like in [15, 1], in the core language only integer terms are used, and the sorts *Heap* and *Addr* are both replaced with *Int*; in case of *Addr*, the range of values is restricted to non-negative numbers by adding explicit domain predicates for all address variables in a formula. The object sort *Object*, and all other ADT sorts, are mapped to integers as in [15].

Our core language provides four predicates specific for heap constraints: $\mathit{heapSize}(h, n)$ expresses that heap h contains n allocated locations; predicate $\mathit{allocHeap}(h, o, h')$ expresses that allocating a fresh address in heap h, and storing object o at this address, yields the new heap h'; $\mathit{read}(h, a, o)$ expresses that

$$
\begin{array}{rcl}
\mathsf{nullAddr} = a & \rightarrowtriangle & 0 = a \\
\mathsf{emptyHeap} = h & \rightarrowtriangle & \mathit{heapSize}(h, 0) \\
\mathsf{allocate}(h, o).\_1 = h' & \rightarrowtriangle & \mathit{allocHeap}(h, o, h') \wedge \\
 & & \exists x.(\mathit{heapSize}(h, x-1) \wedge \mathit{heapSize}(h', x) \wedge \mathit{read}(h', x, o)) \\
\mathsf{allocate}(h, o).\_2 = a & \rightarrowtriangle & \mathit{heapSize}(h, a-1) \\
\mathsf{read}(h, a) = o & \rightarrowtriangle & \mathit{read}(h, a, o) \wedge \\
 & & \exists x.(\mathit{heapSize}(h, x) \wedge ((0 < a \wedge a \leq x) \vee \mathit{defObj} = o))) \\
\mathsf{write}(h, a, o) = h' & \rightarrowtriangle & \mathit{write}(h, a, o, h') \wedge \\
 & & \exists x.(\mathit{heapSize}(h, x) \wedge \mathit{heapSize}(h', x) \wedge \\
 & & \quad (0 \geq a \vee a > x \vee (0 < a \wedge a \leq x \wedge \mathit{read}(h', a, o)))) \\
\mathsf{valid}(h, a) & \rightarrowtriangle & \exists x.(\mathit{heapSize}(h, x) \wedge 0 < a \wedge a \leq x) \\
\neg\mathsf{valid}(h, a) & \rightarrowtriangle & \exists x.(\mathit{heapSize}(h, x) \wedge (0 \geq a \vee a > x)) \\
h \neq h' & \rightarrowtriangle & \exists x, x', a, o, o'. \\
 & & \left(\begin{array}{l} \mathit{heapSize}(h, x) \wedge \mathit{heapSize}(h', x') \wedge \\ \quad (x \neq x' \vee \\ \quad\quad (x = x' \wedge 0 < a \wedge a \leq x \wedge \\ \quad\quad\ \mathit{read}(h, a, o) \wedge \mathit{read}(h', a, o') \wedge o \neq o')) \end{array}\right)
\end{array}
$$

Table 5. Rewriting rules for translation of flat heap formulas to the core language. The rules only apply in positive positions. In the rules, a is an address variable, h, h' are heap variables, and o, o' are heap object variables.

$$
\frac{\Gamma \vdash h_1 = h_2, \Delta \qquad \Gamma, \ldots, t = t' \vdash \Delta}{\Gamma, \mathit{heapSize}(h_1, t), \mathit{heapSize}(h_2, t') \vdash \Delta}\ \text{HEAP-SIZE-FC}
$$

$$
\frac{\Gamma \vdash h_1 = h_2, \Delta \qquad \Gamma, \ldots \vdash a = a', \Delta \qquad \Gamma, \ldots, a = a', o = o' \vdash \Delta}{\Gamma, \mathit{read}(h_1, a, o), \mathit{read}(h_2, a', o') \vdash \Delta}\ \text{READ-FC}
$$

$$
\frac{\Gamma \vdash h_2 = h_3, \Delta \quad \Gamma, \ldots, \mathit{read}(h_1, a', o') \vdash a = a', \Delta \quad \Gamma, \ldots, a = a' \vdash \Delta}{\Gamma, \mathit{write}(h_1, a, o, h_2), \mathit{read}(h_3, a', o') \vdash \Delta}\ \text{ROW}^{\downarrow}
$$

$$
\frac{\Gamma \vdash h_1 = h_3, \Delta \quad \Gamma, \ldots, \mathit{read}(h_2, a', o') \vdash a = a', \Delta \quad \Gamma, \ldots, a = a' \vdash \Delta}{\Gamma, \mathit{write}(h_1, a, o, h_2), \mathit{read}(h_3, a', o') \vdash \Delta}\ \text{ROW}^{\uparrow}
$$

$$
\frac{\Gamma \vdash h_2 = h_3, \Delta \quad \Gamma, \ldots, \mathit{read}(h_1, a', o') \vdash a = a', \Delta \quad \Gamma, \ldots, a = a' \vdash \Delta}{\Gamma, \mathit{allocHeap}(h_1, o, h_2), \mathit{heapSize}(h_2, a), \mathit{read}(h_3, a', o') \vdash \Delta}\ \text{ROA}^{\downarrow}
$$

$$
\frac{\Gamma \vdash h_1 = h_3, \Delta \quad \Gamma, \ldots, \mathit{read}(h_2, a', o') \vdash a = a', \Delta \quad \Gamma, \ldots, a = a' \vdash \Delta}{\Gamma, \mathit{allocHeap}(h_1, o, h_2), \mathit{heapSize}(h_2, a), \mathit{read}(h_3, a', o') \vdash \Delta}\ \text{ROA}^{\uparrow}
$$

Table 6. Sequent calculus rules for heap formulas in the core language. The rules are only applicable if the equation in the first premise follows from equations between heap variables in Γ; i.e., the compared terms are in the same equivalence class constructed by congruence closure. The dots ... stand for the matched literals in the conclusion.

reading from address a in heap h yields object o; and $write(h, a, o, h')$ expresses that storing object o at address a in heap h yields the new heap h'.

The intended semantics of the four predicates essentially follows the heap semantics in Section 2.1, which in particular means that reading from unallocated addresses yields some default value *defObj*, and that writing to unallocated addresses does not change a heap. The null address is, following the standard convention, represented by 0, and the address allocated by $allocHeap(h, a, o, h')$ is the next free address in h, and coincides with the size of the heap h'.

In addition to the four heap predicates, the core language also provides the predicates necessary to represent ADTs; the details of this reduction are given in [15], but the core language is essentially agnostic of the object representation.

3.2 Translation to the Core Language

For sake of presentation, we make several simplifying assumptions when defining the translation of (quantifier-free) heap formulas ϕ to the core language. (i) We assume that ϕ has been brought into a *flat* form upfront. A formula ϕ is flat if function symbols (in particular the functions in Table 1) only occur in equations of the form $g(x_1, \ldots, x_n) = x_0$ (where $x_0, \ldots, x_n$ do not contain functions), and only in positive positions (under an even number of negations). We further assume that (ii) ϕ is in negation normal form, and that (iii) $\mathsf{allocate}$ only occurs in the form $\mathsf{allocate}(\ldots).\_1$ or $\mathsf{allocate}(\ldots).\_2$, i.e., the result of allocation is directly projected to the new heap or the allocated address. Finally, we assume that (iv) the object domain *Object* is infinite, so that the mapping to *Int* defined in [15] does not introduce any side conditions. The assumptions (i)–(iii) can be established by rewriting the considered heap formula. Flatness can be established at the cost of introducing a linear number of additional variables.

Given a formula ϕ satisfying those assumptions, and containing variables $x_1, \ldots, x_l$ with sorts $\sigma_1, \ldots, \sigma_l$, the translation to a formula $\tilde{\phi}$ in the core language is then defined as follows:

$$\tilde{\phi} \quad =_{\text{def}} \quad adtReduction\left(heapRewr(\phi) \wedge \bigwedge_{i=1}^{l} In_{\sigma_i}(x_i)\right) \tag{1}$$

In this definition, *heapRewr* is the function defined by the rewriting rules in Table 5, *adtReduction* is the ADT reduction defined in [15], and $In_\sigma(x)$ are domain constraints defined as:

$$In_\sigma(x) \quad =_{\text{def}} \quad \begin{cases} true & \text{if } \sigma = Heap \\ x \geq 0 & \text{if } \sigma = Addr \\ true & \text{otherwise} \end{cases} \tag{2}$$

Note that, as a slight abuse of notation, the formulas ϕ and $\tilde{\phi}$ contain the same variables $x_1, \ldots, x_l$, we only interpret the variables in $\tilde{\phi}$ as integer variables.

Example 1. Consider the following formula $\phi = A \wedge B$, which is unsatisfiable in the theory of heap storing integers as heap objects:

$$\underbrace{\mathsf{valid}(h, a) \wedge \mathsf{write}(h, a, 42) = h'}_{A} \wedge \underbrace{\mathsf{read}(h', a) = 43}_{B}$$

The formula contains a write that stores 42 at a valid address of h, so that read from the same address of the updated heap h' must return 42. If the rewriting rules are applied to ϕ using the definition from (1), we obtain the following formula $\tilde{\phi}$ in the core language; *adtReduction* has no effect as there are no ADT functions nor predicates:

$$\begin{array}{l} a \geq 0 \wedge \exists x.(\mathit{heapSize}(h, x) \wedge 0 < a \wedge a \leq x) \wedge \\ \left(\begin{array}{l} \mathit{write}(h, a, 42, h') \wedge \exists x.(\mathit{heapSize}(h, x) \wedge \mathit{heapSize}(h', x) \wedge \\ \quad (0 \geq a \vee a > x \vee (0 < a \wedge a \leq x \wedge \mathit{read}(h', a, 42)))) \end{array}\right) \wedge \\ \big(\mathit{read}(h', a, 43) \wedge \exists x.(\mathit{heapSize}(h', x) \wedge ((0 < a \wedge a \leq x) \vee \mathit{defObj} = 43))\big) \end{array} \tag{3}$$

3.3 The Sequent Calculus for the Core Language

Table 6 shows the additional calculus rules (beyond the rules discussed in Section 2.2) needed to reason about heap formulas in the core language: two rules establishing *functional consistency* of the relations *heapSize* and *read*, two rules capturing the *read-over-write* (row) axiom of heaps, and two rules capturing the *read-over-allocation* (roa) axiom. All of the rules have a first premise that asserts the equality of multiple heap terms; such equalities are handled explicitly since the calculus never rewrites predicate literals.

Functional consistency is not needed for the relations *write* and *allocHeap*, since the read-over-write and read-over-allocation rules are sufficient to reason about the heaps produced by those relations. We do not need rules encoding extensionality of heap either, since the only way to observe heap (dis-)equality is through negated equations $h \neq h'$, which are already transformed away by the last rewriting rule in Table 5.

The ROW$^{\downarrow}$ and ROW$^{\uparrow}$ rules can be used to move a *read* literal over a *write* literal, provided that the address a' that is read from is different from the address a written to. The second premise of the rules represents the case that $a \neq a'$, and introduces a new *read* literal; the third premise represents the $a = a'$ case. Interestingly, it is not necessary to check whether the addresses a, a' are valid in the considered heaps, since writing to an invalid address does not mutate a heap.

The rules ROA$^{\downarrow}$ and ROA$^{\uparrow}$ describe the same transformation for the combination of a *read* with an *allocHeap* literal.

Example 2. We continue Example 1, and show how to construct a proof tree for (3). For sake of presentation, we first simplify (3) by introducing Skolem symbols for the quantified variables, and contextual simplification, leading to:

$$\left.\begin{array}{l} \mathit{heapSize}(h, n) \wedge \mathit{heapSize}(h', n) \wedge 0 < a \wedge a \leq n \wedge \\ \mathit{write}(h, a, 42, h') \wedge \mathit{read}(h', a, 42) \wedge \end{array}\right\} A'$$

$$\left.\begin{array}{l} \mathit{read}(h', a, 43) \wedge \mathit{heapSize}(h', n') \wedge ((0 < a \wedge a \leq n') \vee \mathit{defObj} = 43) \end{array}\right\} B'$$

We can then prove unsatisfiability of $A' \wedge B'$ by constructing a proof starting with the sequent $A', B' \vdash \emptyset$. The main step in the proof is the application of the rule READ-FC for functional consistency of *read*:

$$\dfrac{\dfrac{\dfrac{}{\ldots \vdash h' = h'}{}^{*} \quad \dfrac{}{\ldots \vdash a = a}{}^{*} \quad \dfrac{}{a = a, 42 = 43 \vdash}{}^{*}}{read(h', a, 42), read(h', a, 43), \ldots \vdash}\ \text{READ-FC}}{A', B' \vdash \emptyset}\ \wedge\text{-LEFT}^{*} \tag{4}$$

3.4 Properties of the Calculus

The following theorem observes soundness and completeness of our calculus, when applied to a formula that has been rewritten to the core language. In addition, we can observe that systematic application of the rules terminates (in the sense that no new formulas can be added anymore) because the rules in Table 6 do not introduce new terms, and do not remove atoms, and therefore only finitely many atoms will be generated. This implies that the calculus even represents a decision procedure.

Theorem 1. *Suppose ϕ is a heap formula satisfying the assumptions in Section 3.2, and $\tilde{\phi}$ is the corresponding formula in the core language. Then ϕ is unsatisfiable if and only if a closed proof of the sequent $\tilde{\phi} \vdash \emptyset$ exists.*

Proof. "$\Leftarrow$" This is the soundness direction. For a proof by contradiction, assume that ϕ is satisfiable, i.e., there is a variable assignment β satisfying ϕ in the structure defined in Section 2.1. There are then bijections $b_{\text{obj}} : I(\mathit{Object}) \to \mathbb{Z}$ and $b_{\text{hp}} : I(\mathit{Heap}) \to \mathbb{Z}$, and we can construct a solution $((\mathbb{Z}, \tilde{I}), \tilde{\beta})$ of $\tilde{\phi}$:

$$\begin{aligned}
\tilde{I}(\mathit{heapSize}) &= \{(n, |b_{\text{hp}}^{-1}(n)|) \mid n \in \mathbb{Z}\} \\
\tilde{I}(\mathit{allocHeap}) &= \{(n, m, b_{\text{hp}}(b_{\text{hp}}^{-1}(n) +\!\!+ [b_{\text{obj}}^{-1}(m)])) \mid n, m \in \mathbb{Z}\} \\
\tilde{I}(\mathit{read}) &= \{(n, a, b_{\text{obj}}(b_{\text{hp}}^{-1}(n)[a-1])) \mid n \in \mathbb{Z} \text{ and } a \in \{1, \ldots, |b_{\text{hp}}^{-1}(n)|\}\} \cup \\
&\quad \{(n, a, b_{\text{obj}}(\mathit{defObj})) \mid n \in \mathbb{Z} \text{ and } a \notin \{1, \ldots, |b_{\text{hp}}^{-1}(n)|\}\} \\
\tilde{I}(\mathit{write}) &= \left\{ \begin{array}{l} (n, a, o, b_{\text{hp}}(b_{\text{hp}}^{-1}(n)[a-1 \mapsto b_{\text{obj}}^{-1}(o)])) \\ \quad \mid n, o \in \mathbb{Z} \text{ and } a \in \{1, \ldots, |b_{\text{hp}}^{-1}(n)|\} \end{array} \right\} \cup \\
&\quad \{(n, a, o, n) \mid n, o \in \mathbb{Z} \text{ and } a \notin \{1, \ldots, |b_{\text{hp}}^{-1}(n)|\}\} \\
\tilde{\beta}(x) &= b_{\text{obj}}(\beta(x)) \text{ if } x : \mathit{Object} \\
\tilde{\beta}(x) &= b_{\text{hp}}(\beta(x)) \text{ if } x : \mathit{Heap} \\
\tilde{\beta}(x) &= \beta(x) \text{ if } x : \mathit{Addr}
\end{aligned}$$

The translation of other variables and relations is as defined in [15].

By checking the cases in Table 5, we can see that $((\mathbb{Z}, \tilde{I}), \tilde{\beta})$ is indeed a solution of $\tilde{\phi}$, and that the sequent $\tilde{\phi} \vdash$ is therefore counter-satisfiable. It can also be checked that the rules preserve this counter-model: whenever $((\mathbb{Z}, \tilde{I}), \tilde{\beta})$ is

a counter-model of the conclusion of a rule application, it will also be a counter-model of at least one of the premises. This means that no closed proof can exist.

"⇒" This is the completeness direction. Suppose ϕ is a formula so that no closed proof exists for the sequent $\tilde{\phi} \vdash \emptyset$; we show that ϕ is satisfiable. For this, assume that $\mathcal{P}$ is a proof-attempt for $\tilde{\phi} \vdash \emptyset$ with a branch that cannot be closed; that $\Gamma \vdash \Delta$ is the last sequent on that branch; and that the rules from Table 6 (and the rest of the calculus) have been applied exhaustively on the branch. Since systematic application of the rules terminates, only finitely many atoms with the predicates $P = \{\mathit{heapSize}, \mathit{read}, \mathit{write}, \mathit{allocHeap}\}$ can be generated.

We extract a solution of ϕ from $\Gamma \vdash \Delta$. First consider the sub-sequent $\Gamma' \vdash \Delta$ of $\Gamma \vdash \Delta$ that is obtained by removing the P-atoms from Γ; since the calculus from Section 2.2 has been applied exhaustively, a counter-model $(\mathbb{Z}, I'), \beta'$ of $\Gamma' \vdash \Delta$ exists.

Heap variables h can only occur in equations $h = h'$ or as arguments of P-atoms in Γ. Define an equivalence relation $h \simeq h'$ as the reflexive and transitive closure of equations between heap variables in Γ. We write $[h]$ for the class of variable h, and $R([h]) = \{(\mathrm{val}_{\beta'}(a), \mathrm{val}_{\beta'}(o)) \mid \mathit{read}(h', a, o) \in \Gamma, h' \simeq h\}$ for the reads in Γ from $[h]$. Since the rule READ-FC has been applied exhaustively, $R([h])$ will contain at most one value $\mathrm{val}_{\beta'}(o)$ for each address $\mathrm{val}_{\beta'}(a)$, i.e., the data read is consistent.

Whenever $R([h]) \neq \emptyset$, then Γ also contains an atom $\mathit{heapSize}(h', t)$ for some $h' \simeq h$, since the rules in Table 5 are designed in such a way that every *read* is accompanied by *heapSize*, and *heapSize*-atoms also exist for the pre- and the post-heap of every *write* and *allocHeap*. The rule HEAP-SIZE-FC ensures that the elements of $[h]$ are assigned consistent sizes, so that we can set

$$S([h]) = \begin{cases} \mathrm{val}_{\beta'}(t) & \text{if } \mathit{heapSize}(h', t) \in \Gamma \text{ for some } h' \simeq h \\ 0 & \text{otherwise .} \end{cases}$$

As shown in [15], from the counter-model $(\mathbb{Z}, I'), \beta'$ it is possible to reconstruct ADT terms, and we can define a bijection $b_{\mathrm{obj}} : I(\mathit{Object}) \to \mathbb{Z}$ (where I is the interpretation defined in Section 2.1) such that whenever the *Object*-ADT-term t is extracted for $\mathrm{val}_{\beta'}(o)$, for some variable o of sort *Object* in ϕ, then $b_{\mathrm{obj}}(t) = \mathrm{val}_{\beta'}(o)$. We can then define $H([h]) = [t_1, \ldots, t_{S([h])}]$ as the heap represented by $[h]$, with

$$t_i = \begin{cases} b_{\mathrm{obj}}^{-1}(v) & \text{if there is } (i, v) \in R([h]) \\ \mathit{defObj} & \text{otherwise .} \end{cases}$$

The solution of ϕ is the variable assignment β, defined by

$$\beta(x) = \begin{cases} H([x]) & \text{if } x : \mathit{Heap} \\ b_{\mathrm{obj}}^{-1}(\mathrm{val}_{\beta'}(x)) & \text{if } x : \mathit{Object} \\ \beta'(x) & \text{if } x : \mathit{Addr} \\ \beta(x) & \text{(as in [15] for other ADT variables) .} \end{cases}$$

To see that β indeed satisfies ϕ, translate β to a structure $((\mathbb{Z}, \tilde{I}), \tilde{\beta})$ as in "$\Leftarrow$". Because the rules $\text{ROW}^{\downarrow}, \text{ROW}^{\uparrow}, \text{ROA}^{\downarrow}, \text{ROA}^{\uparrow}$ have been applied exhaustively, this structure is a counter-model of $\Gamma \vdash \Delta$. By checking the rules of the calculus individually, we can further see that whenever $((\mathbb{Z}, \tilde{I}), \tilde{\beta})$ is a counter-model of one of the premises of a rule application, it is also a counter-model of the conclusion, and therefore a model of $\tilde{\phi}$. □

4 Craig Interpolation in the Theory of Heap

It is well-known that the (standard) quantifier-free theory of arrays does not admit Craig interpolation: in some cases all interpolants for an unsatisfiable, quantifier-free conjunction $A \wedge B$ will need quantifiers [21]. The same observation applies to the theory of heap. For software verification, however, even imperfect interpolation procedures are useful, and the interesting question arises how the interpolating calculus from Section 2.2 can be generalised to heap formulas.

For interpolation, the conjuncts of an interpolation problem $A \wedge B$ can be translated to the core language independently, i.e., an interpolant $\tilde{I}$ of the rewritten conjunction $\tilde{A} \wedge \tilde{B}$ is computed. Since $\tilde{I}$ will be an interpolant in the core language as well, it has to be mapped back to a normal heap formula I by replacing the relations from Section 3.1 with the original heap functions (Section 2.1). Whether this is possible in all cases is a question that require more research.

For interpolation in the core language, interpolating versions of the heap rules (Table 6) are needed. We follow the approach used in [5, 1] (which in turn resembles the use of theory lemmas in SMT in general), which we summarise in this section. When translating a proof to an interpolating proof, we replace applications of the heap rules with instantiation of an equivalent theory axiom QAx. Suppose a non-interpolating proof contains a rule application

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, \Gamma', \Gamma_1 \vdash \Delta_1, \Delta', \Delta \end{array} \quad \cdots \quad \begin{array}{c}\vdots\\ \Gamma, \Gamma', \Gamma_n \vdash \Delta_n, \Delta', \Delta \end{array}}{\begin{array}{c}\Gamma, \Gamma' \vdash \Delta', \Delta\\ \vdots\end{array}} R$$

in which Γ', Δ' are the formulas assumed by the rule application, Γ, Δ are side formulas not required or affected by the application, and $\Gamma_1, \Delta_1, \ldots, \Gamma_n, \Delta_n$ are newly introduced formulas in the individual branches.

The (unquantified) theory axiom Ax corresponding to the rule application expresses that the conjunction of the premises has to imply the conclusion; the quantified theory axiom $QAx =_{\text{def}} \forall S.\, Ax$ in addition contains universal quantifiers for all variables S occurring in Ax.

$$Ax \;=_{\text{def}}\; \bigwedge_{i=1}^{n} \left(\bigwedge \Gamma_i \to \bigvee \Delta_i\right) \;\to\; \left(\bigwedge \Gamma' \to \bigvee \Delta'\right)$$

Ax and QAx are specific to the *application* of R: the axioms for two distinct applications of R will in general be different formulas. QAx is defined in such

a way that the effect of R can be simulated by introducing QAx in the antecedent, instantiating it with the right terms, and applying propositional rules. This construction leads to a proof using only the standard rules from Section 2.2, which can be interpolated as discussed earlier. Since QAx is a valid formula not containing any free variables, it can be introduced in a proof at any point, and labelled $\lfloor QAx \rfloor_L$ or $\lfloor QAx \rfloor_R$ on demand.

An immediate consequence of this approach is the possibility of quantifiers occurring in interpolants. This is because the interpolating rules $\forall$-LEFT$_{L/R}$ (Table 4) have to introduce quantifiers $\forall_{Rt}/\exists_{Lt}$ for local symbols in the substituted term t; whether such quantifiers actually occur in the final interpolant depends on the applied heap rules, and on the order of rule application. However, as we have observed in the beginning of the section, quantifiers in heap interpolants are in general unavoidable.

Example 3. We continue Example 2, and show how to extract an interpolant for the conjunction $A \wedge B$. Since A' and B' in Example 2 are simplified versions of the conjuncts $\tilde{A}$ and $\tilde{B}$ in the core language, respectively, interpolation can start A', B'. The rule application READ-FC can be encoded as the following theory axioms:

$$\begin{aligned} Ax &= (h' = h' \wedge a = a \wedge 42 \neq 43) \rightarrow \neg(read(h', a, 42) \wedge read(h', a, 43)) \\ &\equiv \neg read(h', a, 42) \vee \neg read(h', a, 43) \\ QAx &\equiv \forall h.\, (\neg read(h, a, 42) \vee \neg read(h, a, 43)) \end{aligned}$$

Replacing the application of READ-FC with the axiom, and adding formula labels, we obtain the following interpolating proof of the conjunction $A' \wedge B'$:

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{\lfloor read(h', a, 42)\rfloor_L \vdash \lfloor read(h', a, 42)\rfloor_R \blacktriangleright \tilde{J}}\,\text{CLOSE}_{LR}}{\lfloor read(h', a, 42)\rfloor_L, \lfloor \neg read(h', a, 42)\rfloor_R \vdash \blacktriangleright \tilde{J}}\,\neg\text{-LEFT}_R \qquad \vdots}{\lfloor read(h', a, 42)\rfloor_L, \lfloor read(h', a, 43)\rfloor_R, \lfloor \neg read(h', a, 42) \vee \neg read(h', a, 43)\rfloor_R \vdash \blacktriangleright \tilde{J} \wedge true}\,\vee\text{-LEFT}_R}{\lfloor read(h', a, 42)\rfloor_L, \lfloor read(h', a, 43)\rfloor_R, \lfloor QAx\rfloor_R, \ldots \vdash \blacktriangleright \tilde{I}}\,\forall\text{-LEFT}_R}{\lfloor A'\rfloor_L, \lfloor B'\rfloor_R \vdash \emptyset \blacktriangleright \tilde{I}}\,\wedge\text{-LEFT}^*$$

The interpolant extracted from this proof is $\tilde{I} \equiv read(h', a, 42)$, and stems from the application of the rule CLOSE$_{LR}$ in the left sub-proof. In the right sub-proof (not shown here), only R-labelled formulas are needed, and the sub-interpolant generated by the rule CLOSE-RR is *true*. The interpolant $\tilde{I}$ in the core language can be translated back to the heap formula $I = (\mathsf{read}(h', a) = 42)$. Note that this formula is a correct interpolant even though it does not explicitly state that a is a valid address in h'.

The proof could also be rewritten to use the L-labelled axiom $\lfloor QAx \rfloor_L$ instead of $\lfloor QAx \rfloor_R$. As a result, the label of several formulas and rule applications would then change from R to L, and the final interpolant becomes $\mathsf{read}(h', a) \neq 43$.

5 Related Work

This paper presents the first (and largely unoptimised) decision and interpolation procedures for the theory of heap, which we hope will facilitate further research. Since the theory of heap is quite close to the theory of arrays, we discuss some existing work on array decision and interpolation procedures in this section.

There is a large body of research on array decision procedures for SMT. Stump et al. present a decision procedure for the extensional theory of arrays, including several extensions [30]. Our rules for heap have similarities with this procedure. De Moura et. al. define a decision procedure for combinatory array logic [24]. Hoenicke et al. present an algorithm for the theory of arrays where lemmas are created lazily based on weak equivalences [9]. Brummayer et al. present a decision procedure for the extensional theory of arrays that introduces lemmas lazily, guided by congruence closure [7].

Interpolation procedures for arrays have been presented in a number of recent publications, in particular tackling the problem of defining array theories that admit quantifier-free interpolation. Bruttomesso et. al. observe that adding a *diff* function to the theory of arrays establishes quantifier-free interpolation, and present an interpolation procedure [8]. An interpolation procedure based on weak equivalences, extending [9], is given in [14], again ensuring quantifier-free interpolants by adding a *diff* function. Totla et. al. present an interpolation procedure for arrays based on complete instantiation [32].

6 Conclusions and Outlook

The paper presents a calculus for deciding heap theory formulas and proves it sound and complete. A procedure to generate interpolants using only the standard rules from Section 2.2 is also presented. The procedures are intended as a starting point to initiate further research on more optimised decision and interpolation procedures for the theory of heap. In particular, we believe that many of the approaches surveyed in the previous section, developed for the theory of arrays, can be adapted also to the theory of heap.

An orthogonal line of research concerns simplification techniques for CHCs modulo the theory of heap. Such techniques can for instance use Abstract Interpretation to derive the validity of heap addresses, or the type of objects at specific addresses. CHCs could also be simplified by eliminating repeated reads from the same address, across multiple clauses, or by additional arguments to predicates in order to partially expand heap arguments.

Acknowledgements. This work was supported by the Swedish Research Council (VR) under grant 2018-04727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

References

1. Backeman, P., Rümmer, P., Zeljic, A.: Bit-vector interpolation and quantifier elimination by lazy reduction. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–10. IEEE (2018), https://doi.org/10.23919/FMCAD.2018.8603023
2. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. JSAT 3(1-2), 21–46 (2007)
3. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. Lecture Notes in Computer Science, vol. 9300, pp. 24–51. Springer (2015), https://doi.org/10.1007/978-3-319-23534-9_2
4. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 105–125. Springer (2013), https://doi.org/10.1007/978-3-642-38856-9_8
5. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In: VMCAI. pp. 88–102. LNCS, Springer (2011)
6. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. Journal of Automated Reasoning 47, 341–367 (2011)
7. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. J. Satisf. Boolean Model. Comput. 6(1-3), 165–201 (2009), https://doi.org/10.3233/sat190067
8. Bruttomesso, R., Ghilardi, S., Ranise, S.: Quantifier-free interpolation of a theory of arrays. Log. Methods Comput. Sci. 8(2) (2012), https://doi.org/10.2168/LMCS-8(2:4)2012
9. Christ, J., Hoenicke, J.: Weakly equivalent arrays. In: Rümmer, P., Wintersteiger, C.M. (eds.) Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014, affiliated with the 26th International Conference on Computer Aided Verification (CAV 2014), the 7th International Joint Conference on Automated Reasoning (IJCAR 2014), and the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014), Vienna, Austria, July 17-18, 2014. CEUR Workshop Proceedings, vol. 1163, pp. 39–49. CEUR-WS.org (2014), http://ceur-ws.org/Vol-1163/paper-06.pdf
10. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic 22(3), 250–268 (September 1957)
11. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Program verification using constraint handling rules and array constraint generalizations. Fundam. Inform. 150(1), 73–117 (2017), https://doi.org/10.3233/FI-2017-1461
12. Esen, Z., Rümmer, P.: Towards an smt-lib theory of heap. In: Fribourg, L., Heizmann, M. (eds.) Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and Synthesis, VPT/HCVS@ETAPS 2020 2020, and 7th Workshop on Horn Clauses for Verification and SynthesisDublin, Ireland, 25-26th April 2020. EPTCS, vol. 320 (2020)

13. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. Springer-Verlag, New York, 2nd edn. (1996)
14. Hoenicke, J., Schindler, T.: Efficient interpolation for the theory of arrays. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10900, pp. 549–565. Springer (2018), https://doi.org/10.1007/978-3-319-94205-6_36
15. Hojjat, H., Rümmer, P.: Deciding and interpolating algebraic data types by reduction. In: Jebelean, T., Negru, V., Petcu, D., Zaharie, D., Ida, T., Watt, S.M. (eds.) 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017. pp. 145–152. IEEE Computer Society (2017), https://doi.org/10.1109/SYNASC.2017.00033
16. Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–7. IEEE (2018), https://doi.org/10.23919/FMCAD.2018.8603013
17. Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017), https://easychair.org/publications/paper/Pmh
18. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: SIGSOFT'06/FSE-14. pp. 105–116. ACM, New York, NY, USA (2006)
19. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 89–96. IEEE (2015)
20. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 846–862. Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_59
21. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 16–30. Springer (2004), https://doi.org/10.1007/978-3-540-24730-2_2
22. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free Horn clauses. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 361–382. Springer (2016), https://doi.org/10.1007/978-3-662-53413-7_18
23. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest,

Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
24. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA. pp. 45–52. IEEE (2009), https://doi.org/10.1109/FMCAD.2009.5351142
25. Reynolds, A., Blanchette, J.C.: A decision procedure for (co)datatypes in SMT solvers. J. Autom. Reasoning 58(3), 341–362 (2017), https://doi.org/10.1007/s10817-016-9372-6
26. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 159–169. ACM (2008), https://doi.org/10.1145/1375581.1375602
27. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNCS, vol. 5330, pp. 274–289. Springer (2008)
28. Rümmer, P.: Competition report: CHC-COMP-20. In: Fribourg, L., Heizmann, M. (eds.) Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and Synthesis, VPT/HCVS@ETAPS 2020 2020, and 7th Workshop on Horn Clauses for Verification and SynthesisDublin, Ireland, 25-26th April 2020. EPTCS, vol. 320, pp. 197–219 (2020), https://doi.org/10.4204/EPTCS.320.15
29. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-clause verification. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 347–363. Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_24
30. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 29–37. IEEE Computer Society (2001), https://doi.org/10.1109/LICS.2001.932480
31. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. SIGPLAN Not. 45(1), 199–210 (Jan 2010)
32. Totla, N., Wies, T.: Complete instantiation-based interpolation. J. Autom. Reason. 57(1), 37–65 (2016), https://doi.org/10.1007/s10817-016-9371-7