An SMT-LIB Theory of Heaps

Zafer Esen[1], Philipp Rümmer[1,2]

[1]Uppsala University, Sweden
[2]University of Regensburg, Germany

Abstract

Constrained Horn Clauses (CHCs) are an intermediate program representation that can be generated by several verification tools, and that can be processed and solved by a number of Horn solvers. One of the main challenges when using CHCs in verification is the encoding of *heap-allocated data-structures:* such data-structures are today either represented explicitly using the theory of arrays, or transformed away with the help of invariants or refinement types, defeating the purpose of CHCs as a representation that is language-independent as well as agnostic of the algorithm implemented by the Horn solver. This paper presents an *SMT-LIB theory of heaps* tailored to CHCs, with the goal of enabling a standard interchange format for programs with heap data-structures. We introduce the syntax of the theory of heaps, define its semantics in terms of axioms and using a reduction to SMT-LIB arrays and data-types, provide an experimental evaluation and outline possible extensions and future work.

Keywords

SMT, Constrained Horn Clauses, Theory of Heaps, SMT-LIB

1. Introduction

Constrained Horn Clauses (CHCs) are a convenient intermediate verification language that can be generated by several verification tools in many settings, ranging from verification of smart contracts [1] to verification of computer programs in various languages [2, 3, 4, 5, 6]. The CHC interchange language provides a separation of concerns, allowing the designers of verification systems to focus on high-level aspects like the applied proof rules and verification methodology, while giving CHC solver developers a clean framework that can be instantiated using various model checking algorithms and specialised decision procedures. Solver performance is evaluated in the annually held CHC-COMP [7].

CHCs are usually expressed using the SMT-LIB standard, which itself is a common language and interface for SMT solvers [8]. Abstractly, both SMT solvers and CHC solvers are tools that determine if a first-order formula is satisfiable modulo background theories such as arithmetic, bit-vectors, or arrays.

One of the main challenges when using CHCs, and in verification in general, is the encoding of programs with mutable, heap-allocated data-structures. Since there is no native theory of heaps in SMT-LIB, one approach to represent such data-structures is using the theory of arrays (e.g., [9, 10]). This is a natural encoding since a heap can be seen as an array of memory locations;

SMT 2022: Satisfiability Modulo Theories, August 11–12, 2022, Haifa, Israel
0000-0002-1522-6673 (Z. Esen); 0000-0002-2733-7098 (P. Rümmer)

however, as the encoding is byte-precise, in the context of CHCs it tends to be low-level and often yields clauses that are hard to solve.

An alternative approach is to transform away such data-structures with the help of invariants or refinement types (e.g., [11, 12, 13, 4]). In contrast to approaches that use the theory of arrays, the resulting CHCs tend to be over-approximate (i.e., can lead to false positives), even with smart refinement strategies that aim at increasing precision. This is because every operation that reads, writes, or allocates a heap object is replaced with assertions and assumptions about local object invariants, so that global program invariants might not be expressible. In cases where local invariants are sufficient, however, they can enable efficient and modular verification even of challenging programs.

Both approaches leave little design choice with respect to handling of heap to CHC solvers. Dealing with heap at encoding level implies repeated effort when designing verifiers for different programming languages, makes it hard to compare different approaches to encode heap, and is time-consuming when a verifier wants to switch to another encoding. The benefits of CHCs are partly negated, since the discussed separation of concerns does not carry over to heap.

The vision of this paper is to extend CHCs to a standardised interchange format for programs with heap data-structures. To this end, we present a high-level theory of heaps that does not restrict the way in which CHC solvers approach heap, while covering the main functionality of heap needed for program verification: (i) representation of the type system associated with heap data; (ii) reading and updating of data on the heap; (iii) handling of object allocation.

We use algebraic data-types (ADTs), as already standardised by SMT-LIB v2.6, as a flexible way to handle (i). The theory offers operations akin to the theory of arrays to handle (ii) and (iii). The theory is deliberately kept simple, so that it is easy to add support to SMT and CHC solvers: a solver can, for instance, internally encode heap using the existing theory of arrays (we provide one such encoding in [14]), or implement transformational approaches like [12, 13]. Since we want to stay high-level, arithmetic operations on pointers are excluded in our theory, as are low-level tricks like extracting individual bytes from bigger pieces of data through pointer manipulation. Being language-agnostic, the theory of heaps allows for common encodings across different applications, and is in the spirit of both CHCs and SMT-LIB.

Contributions of the paper are (i) the definition of syntax and semantics of the theory of heaps, (ii) a collection of an initial set of benchmarks, (iii) experimental results.

Acknowledgements This is the first full paper introducing the theory of heaps (a detailed account of an earlier version of the theory is available as a technical report [14]). Earlier versions of the theory were presented at the HCVS Workshop 2020 [15] and the SMT Workshop 2020 [16]. An invited paper at LOPSTR 2020 discusses preliminary work on decision and interpolation procedures [17]. We are grateful for the discussion and feedback provided by the different communities. This work was supported by the Swedish Research Council (VR) under grant 2018-04727, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and by the Knut and Alice Wallenberg Foundation under the project UPDATE.

2. Motivating Example

We start with a high-level explanation how heap is handled by our theory. Listing 1 shows

Listing 1
The motivating example in Java

```
1  abstract class IntList {
2    protected int _sz;
3    abstract int hd();
4    abstract void setHd(int hd);
5    abstract IntList tl();
6    int sz() {return _sz;} }
7
8  class Nil extends IntList {
9    Nil() {_sz = 0;}
10   int hd () {err();}
11   void setHd (int hd) {err();}
12   IntList tl () {err();} }
13
14 class Cons extends IntList {
15   int _hd;
16   IntList _tl;
17   int hd() {return _hd;}
18   void setHd (int hd) {_hd=hd;}
19   IntList tl() { return _tl; }
20   Cons(int hd, IntList tl) {
21     _hd = hd;
22     _tl = tl;
23     _sz = 1 + tl.sz(); } }
24 class Motivation {
25   void main() {
26     IntList l = new Cons(42,
27                          new Nil());
28     l.setHd(l.hd()+1);
29     assert(l.hd() == 43);
30   }
31 }
```

a simple Java program that constructs a singly-linked list through heap operations such as allocation on the heap (lines 26–27), reading (lines 28–29) and modifying (line 28) heap data.

In order to encode this program we use constrained Horn clauses (CHCs). We refer to sources such as [18, 2] for a comprehensive explanation of using CHCs in this context. Although the theory of heaps is presented in the context of CHCs, there is nothing CHC-specific in the theory itself; the theory can be supported by both SMT and CHC solvers since it is kept deliberately high level and simple. The encoding is given in Listing 2 in SMT-LIB v2.6 format.

Heap declaration To encode this program using the theory of heaps, first a heap has to be declared that covers the program types as shown at lines 1–12 of Listing 2. Each heap comes with its own sorts for the heap itself and for heap locations (or addresses). Lines 2 and 3 are the names of declared heap and address sorts. We next need to define which data can be placed on the heap, which is done by choosing the sort of heap objects; this sort can be any of the sorts declared prior to or together with the heap declaration, excluding the heap sort itself. Line 4 specifies the object sort to be the ADT `Object`, declared later.

Line 5 defines the object assumed to be stored at *unallocated* heap locations. Since functions in SMT-LIB are total, semantics has to be defined also for reads from such unallocated addresses. The theory of heaps leaves the choice of object produced by such reads to the user; the term specified at line 5 must have the object sort chosen at line 4. We call this the default object (or $defObj$), which in this case is created using the object constructor `O_Empty`. There are two main reasons why the result of a read from an unallocated location is not left unspecified: 1. the axioms become more complicated (for instance [row2] from Table 2 would require limiting p_2 to allocated addresses); 2. deallocation semantics can be partly achieved (see Section 4).

The rest of the heap declaration at lines 6–12 corresponds to an SMT-LIB data-type declaration. In line 6, in addition to `Object` we declare data-types `IntList`, `Cons`, and `Nil`, encoding the classes of the program. The constructors at lines 7–9 specify the fields of each class, and in addition give `Cons` and `Nil` each a field containing the parent `IntList` object. In lines 10–12,

the constructors of the `Object` sort are declared, which correspond to the classes `Cons` and `Nil`, as well as the default object `O_Empty`. The class `IntList` is abstract and does not occur directly on the heap, so that no constructor for this type is provided.

Since each heap theory has its own address sort, cases are immediately prevented in which multiple heaps share the same address sort, or in which some other interpreted sort (say, `Int`) is used to store addresses. This rules out accidental cases of pointer arithmetic, and leaves full flexibility to solvers on how to internally represent addresses (e.g., see [4]). This choice also implies that ADT declarations for heap objects that refer to address sorts need to be part of `declare-heap`.

Within one heap, all pointers are represented using a single *Addr* sort. No distinction is made between pointers to objects from different constructors. This is close in semantics to languages like C, where casts between arbitrary pointer types are possible and it has to be verified for each heap access that indeed an object of the right type is accessed. In languages like Java, the

Listing 2: SMT-LIB encoding of the motivating example from Listing 1. The symbols of some sorts and operations of the theory are abbreviated and the list of quantified variables is skipped in some cases for brevity.

```
(declare-heap
 Heap                                      ; declared Heap sort
 Addr                                      ; declared Address sort
 Object                                    ; chosen Object sort
 O_Empty                                   ; the default Object
 ((IntList 0) (Cons 0) (Nil 0) (Object 0)) ; ADTs
 (((IntList   (sz         Int)))           ; Class constructors
  ((Cons      (parentCons IntList) (hd Int) (tl Addr)))
  ((Nil       (parentNil  IntList)))
  ((O_Cons    (getCons    Cons))           ; Object sort constructors
   (O_Nil     (getNil     Nil))
   (O_Empty   ))))
                                           ; invariant declarations
(declare-fun I1 (Heap)       Bool)         ; <h>
(declare-fun I2 (Heap Addr) Bool)          ; <h,p>
(declare-fun I3 (Heap Addr) Bool)          ; <h,l>
(declare-fun I4 (Heap Addr) Bool)          ; <h,l>

(assert (I1 emptyHeap))
(assert (forall ((h Heap) (h1 Heap) (p1 Addr))
  (=> (and (I1 h) (= (ARHeap h1 p1) (alloc h (O_Nil (Nil (IntList 0))))))
      (I2 h1 p1))))
(assert (forall (...)
  (=> (and (I2 h p)
           (= (ARHeap h1 p1) (alloc h (O_Cons (Cons (IntList 1) 42 p)))))
      (I3 h1 p1))))
(assert (forall (...)
  (=> (and (I3 h l) (not (valid h l))) false)))
(assert (forall ((pn IntList) (head Int) (tail Addr) ...)
  (=> (and (I3 h l) (= h1 (write h l (O_Cons (Cons pn (+ 1 head) tail))))
           (= (O_Cons (Cons pn head tail)) (read h l))) (I4 h1 l))))
(assert (forall (...)
  (=> (and (I3 h l) (= (O_Nil (Nil pn)) (read h l))) false)))
(assert (forall (...)
  (=> (and (I4 h l) (= (O_Cons (Cons pn head tail)) (read h l))
           (not (= head 43))) false)))
(assert (forall (...)
  (=> (and (I4 h l) (not (is-O_Cons (read h l)))) false)))
```

stronger type system will provide information about the objects a variable can refer to, but exceptions can be raised when performing casts. The theory of heaps is flexible enough to cover those different settings.

Apart from the sorts mentioned, the heap declaration implicitly declares (among others) an ADT ARHeap (also called $AllocResultHeap$ later in the paper) that holds pairs $\langle Heap, Addr \rangle$ returned as a result of allocations.

Program encoding Predicates representing program states are declared at lines 14–17. The first set of arguments in the parentheses lists the sorts of the variables we want to keep track of at that point. E.g., for line 17, we want to have a global view of the heap, as well as all variables on the stack at that point. The only variable on the stack at this point is a temporary variable p that corresponds to the newly allocated `Nil` object's address (line 27 in Listing 1).

Line 19 is the program entry point, where the heap is initially empty. The function emptyHeap returns an empty heap (i.e., unallocated at all locations) of the declared *Heap* sort specified at line 2. Lines 20–26 allocate, respectively, a `Nil` object and a `Cons` object on the heap. Allocation is done using the alloc function of the theory, which takes as arguments the old heap and the new object to be put on the heap, and returns an ARHeap pair with the new heap and the allocated address. Constructor calls are inlined and slightly simplified in the encoding. For example, line 25 shows the simplified encoding of the Java constructor for `Cons` at lines 20–23 of Listing 1. The update of the `_sz` field is simplified by directly assigning a value to it, which would actually require another clause with a read due to the statement at line 23.

Lines 27–33 illustrate the use of read and write functions. read reads from the provided heap at the given location, and write writes the provided object to the heap at the specified location. The assertion at lines 27–28 checks the validity of accesses in order to ensure memory safety. The dynamic dispatch needed when calling `hd` is implemented through pattern matching using the `O_Cons` and `O_Nil` constructors: in lines 29–31 the method call is successful, and the heap object is subsequently updated, while the clause at lines 32–33 models the error when executing `Nil.hd`. The same property can be expressed using the tester `is-O_Cons` in lines 37–38. Lastly, lines 34–36 encode the assertion at line 29 from Listing 2.

3. Vocabulary and Syntax of the Theory of Heaps

3.1. SMT-LIB-style Declaration of Heaps

A theory of heaps is declared as follows:

$$\boxed{(\texttt{declare-heap}\ c_h\ c_a\ c_o\ \tau_o\ ((\delta_1 k_1)\ ...\ (\delta_n k_n))\ (d_1 ... d_n))}$$

where $c_h,\ c_a,\ c_o$ are symbols corresponding to the names of declared heap, declared address and chosen object respectively. τ_o is a term of the chosen object which is returned on invalid accesses (i.e. the default object). The object sort can be chosen as any sort without c_h in its signature. The rest of the declaration resembles the `declare-datatypes` declaration from the SMT-LIB standard v2.6 [8], with the exception that polymorphism is (currently) not supported in constructor declarations (a discussion is provided in [14]), and that there should be n (where

Table 1
Operations defined by the theory of heaps

emptyHeap :	()	$\rightarrow$ *Heap*	(1)
nullAddr :	()	$\rightarrow$ *Addr*	(2)
alloc :	*Heap* $\times$ *Object*	$\rightarrow$ *Heap* $\times$ *Addr*	(3)
valid :	*Heap* $\times$ *Addr*	$\rightarrow$ *Bool*	(4)
read :	*Heap* $\times$ *Addr*	$\rightarrow$ *Object*	(5)
write :	*Heap* $\times$ *Addr* $\times$ *Object*	$\rightarrow$ *Heap*	(6)

$n \geq 0$) instead of $n+1$ ADT sort declarations (i.e., the object sort can also be declared before the heap declaration and specified using c_o, if it does not use the address sort (c_a) in its declaration).

The concrete syntax for the heap declaration is given below, which extends $\langle command \rangle$ in the concrete syntax of SMT-LIB v2.6.

$\langle command \rangle$ ::= ...
| (`declare-heap` $\langle symbol \rangle$ $\langle symbol \rangle$ $\langle sort \rangle$ $\langle term \rangle$
($\langle sort\_dec \rangle^n$) ($\langle heap\_datatype\_dec \rangle^n$))

$\langle heap\_datatype\_dec \rangle$ ::= $\langle constructor\_dec \rangle^+$

The first two symbols and the following sort in the declaration correspond respectively to c_h, c_a and c_o from the abstract syntax. $\langle term \rangle$ is the default object.

3.2. Sorts

Each heap declaration introduces several sorts: 1. a sort *Heap* of heaps, 2. a sort *Addr* of heap addresses, 3. zero or more ADT sorts used to represent heap data, 4. an additional ADT sort that holds the pair $\langle Heap, Addr \rangle$ which is the result of calling alloc. In order to make this ADT sort distinguishable, it is suffixed with its associated heap sort *Heap* (e.g. *AllocResultHeap*). The names of these sorts are defined by the variables in the `declare-heap` command, which we assume in this paper to be *Heap* for c_h and *Addr* for c_a.

3.3. Operations and Semantics

A list of operations of the theory of heaps is given in Table 1. Although not listed in the table, we also assume access to all ADT operations for the ADTs declared by the theory of heaps. Some operations contain the symbols `Heap` and `Addr` in their signatures. This is done with the assumption that the declared heap and address sorts are named *Heap* and *Addr* respectively. E.g., `nullAddress` would be `nullA` if the declared address sort was named *A*, and it would return a value of sort *A*. Including the sort name in some function and sort names makes it possible to determine their associated heap declarations without using the SMT-LIB command "`as`". This is not required in sorts and operations where the associated heap sort is clear, such as in read (its first argument is of heap sort).

A set of axioms formalising the semantics of the theory of heaps is given in Table 2. In addition, a definition of the axioms in terms of the theory of arrays is provided in [14]. Below we provide an informal description for each operation of the theory.

Function alloc takes a $Heap$ and an $Object$, and returns a data-type $AllocResultHeap$ representing the pair $\langle Heap, Addr \rangle$. The returned $Heap$ at $Addr$ contains the passed $Object$, with all other locations unchanged. The pair ADT is required as the return sort since it is not possible in SMT-LIB to return the two values separately. In Section 4 we discuss other alternatives such as using multiple allocation functions.

Functions read and write are similar to the array *select* and *store* operations [19]; however, unlike an array, a heap also carries information about allocatedness. The predicate valid checks if accesses to a given $Heap$ at a given $Addr$ are valid. We say that an access is *valid* if and only if that location was allocated beforehand by using the function alloc, and *invalid* otherwise. The function nullAddr returns the $Addr$ that is always unallocated and emptyHeap returns the $Heap$ that is unallocated at all locations.

The functions read and write behave as their array counterparts if the access is valid. Invalid reads return a default $Object$ to make the function total (as explained in Section 2). The write function returns a new $Heap$ if the access is valid, otherwise the original $Heap$ is returned without any changes. Validity of a write can be independently checked via memory-safety assertions as shown in lines 27–28 of Listing 2.

We propose a further short-hand notation $\mathsf{nthAddr}_i$, which is useful when presenting satisfying assignments. It is used to concisely represent $Addr$ values which would be returned after i alloc calls. This short-hand notation is only possible with the deterministic allocation axiom [alloc2] given in Table 2.

The properties of the theory of heaps are given in [14]. In particular, satisfiability of quantifier-free heap formulas is NP-complete (provided that the theory chosen to represent heap objects is by itself in NP). Like for arrays, NP-completeness can be observed already for conjunctions of heap literals.

4. Alternative Definitions and Extensions

This section explains the rationale behind some of the design choices in the theory of heaps, as well as some natural extensions. It is intended as a starting point for further discussions and a standardisation within SMT-LIB.

$AllocResultHeap$ Allocation on the heap needs to produce both a new heap and a fresh address. In our theory, the pair of new heap and new address is handled using the ADT $AllocResultHeap$, which enables us to stick to just a single allocation function alloc. Alternatively, alloc could be represented using a pair of functions, choosing for instance $\mathsf{alloc}(h, o) = \langle \mathsf{allocHeap}(h, o), \mathsf{allocAdress}(h, o) \rangle$; this would be preferable from a solver implementation point of view, but not necessarily for users. Altogether this point is more of aesthetic concern.

Deterministic allocation In its current semantics, object allocation in the theory of heaps is deterministic: since alloc is a function, it will always produce the same fresh address when

Table 2
Axiomatic semantics of the theory of heaps. All variables occurring in the axioms are universally quantified with sorts $h : Heap$, $p : Addr$, $r : AddrRange$, $o : Object$ and $ar : AllocResultHeap$. Variables can appear subscripted. $AllocResultHeap$ is a pair $\langle Heap, Addr\rangle$; we use the notation $ar.\_1$ and $ar.\_2$ to select the pair's fields.

Read over write	$\mathsf{valid}(h, p) \rightarrow \mathsf{read}(\mathsf{write}(h, p, o), p) = o$	[row1]
	$p_1 \neq p_2 \rightarrow \mathsf{read}(\mathsf{write}(h, p_1, o), p_2) = \mathsf{read}(h, p_2)$	[row2]
Read over allocate	$\mathsf{alloc}(h, o) = ar \rightarrow \mathsf{read}(ar.\_1, ar.\_2) = o$	[roa1]
	$\mathsf{alloc}(h, o) = ar \ \wedge \ p \neq ar.\_2 \rightarrow \mathsf{read}(ar.\_1, p) = \mathsf{read}(h, p)$	[roa2]
Allocation	$\mathsf{alloc}(h, o) = ar \rightarrow \neg\mathsf{valid}(h, ar.\_2) \wedge \mathsf{valid}(ar.\_1, ar.\_2) \wedge (\forall p : Addr.(ar.\_2 \neq p \rightarrow (\mathsf{valid}(h, p) \leftrightarrow \mathsf{valid}(ar.\_1, p))))$	[alloc1]
	$(\forall p : Addr.(\mathsf{valid}(h_1, p) \leftrightarrow \mathsf{valid}(h_2, p))) \rightarrow \mathsf{alloc}(h_1, o_1).\_2 = \mathsf{alloc}(h_2, o_2).\_2$	[alloc2]
Invalid access	$\neg\mathsf{valid}(h, p) \rightarrow \mathsf{write}(h, p, o) = h$	[ivwt]
	$\neg\mathsf{valid}(h, p) \rightarrow \mathsf{read}(h, p) = \ defObj$	[ivrd]
Validity	$\neg\mathsf{valid}(\mathsf{emptyHeap}, p)$	[vld1]
	$\neg\mathsf{valid}(h, \mathsf{nullAddr})$	[vld2]
	$h_2 = \mathsf{write}(h_1, p_1, o_1) \rightarrow (\mathsf{valid}(h_1, p_2) \leftrightarrow \mathsf{valid}(h_2, p_2))$	[vld3]
Extensionality	$(\forall p : Addr.(\mathsf{valid}(h_1, p) \leftrightarrow \mathsf{valid}(h_2, p)) \wedge \mathsf{read}(h_1, p) = \mathsf{read}(h_2, p)) \rightarrow h_1 = h_2$	[ext]
No-junk (constructability)	$\exists f : \mathbb{N} \rightarrow Heap, g : \mathbb{N} \rightarrow Addr. f(0) = \mathsf{emptyHeap} \wedge g(0) = \mathsf{nullAddr} \wedge \forall i : \mathbb{N}.\ \langle f(i+1), g(i+1)\rangle = \mathsf{alloc}(f(i), defObj) \wedge \forall p : Addr.\ \exists i : \mathbb{N}.\ g(i) = p$	[cons]

applied to the same arguments. Moreover, [alloc2] implies that the new address is determined entirely by the set of already allocated addresses on the heap. Determinism simplifies the presentation of models and counterexamples through the function nthAddr. Determinism also simplifies the computation of program invariants, since it implies the existence of a linear order of the heap addresses (as witnessed by the array semantics discussed in [14]): an invariant can distinguish fresh and used addresses using a simple inequality. Determinism will in many practical cases not be observable in programs: the syntax of the theory of heaps prevents arithmetic on addresses, and normal program semantics does not allow alloc to be called repeatedly on the same heap in any case.

In cases where it is needed, there is an elegant way to reintroduce non-determinism: the alloc function can be given a third argument (nonce/entropy), as in $\mathsf{alloc}(h, o, e)$, and the axiom [cons] be relativised to only hold for fixed values of e. The axiom [alloc2] could be dropped. The translation of programs to CHCs can then choose a non-deterministic value for e when encoding an allocation operation like new. A side effect of this change would be that decision procedures and correct encoding of heaps using arrays become more complex, and for instance

Table 3
Extended operations over address ranges

$$\begin{aligned}
\mathsf{batchAlloc} &: \mathit{Heap} \times \mathit{Object} \times \mathbb{N} \rightarrow \mathit{Heap} \times \mathit{AddrRange} && (7)\\
\mathsf{batchWrite} &: \mathit{Heap} \times \mathit{AddrRange} \times \mathit{Object} \rightarrow \mathit{Heap} && (8)\\
\mathsf{nthInRange} &: \mathit{AddrRange} \times \mathbb{N} \rightarrow \mathit{Addr} && (9)\\
\mathsf{contains} &: \mathit{AddrRange} \times \mathit{Addr} \rightarrow \mathit{Bool} && (10)\\
\mathsf{validRange} &: \mathit{Heap} \times \mathit{AddrRange} \rightarrow \mathit{Bool} && (11)
\end{aligned}$$

have to store the allocation status of each address using a bit-array.

Deallocation A natural extension of the theory is the addition of a function for deallocating objects, which would be helpful to capture languages without garbage collection like C/C++; for such languages deallocation otherwise has to be encoded using an explicit flag added to objects. The effect on the theory semantics would be similar as for non-deterministic allocation: decision procedures would need to maintain a bit-array to remember the allocatedness of addresses.

The theory of heaps already provides a way to obtain partial deallocation semantics without the additional bit-array. If the default object sort is chosen to be one of the sorts not corresponding to any program type, then the semantics of deallocation in most programming languages can be achieved by writing back the default object to deallocated locations. Then valid heap/address pairs that return the default object on a read imply that they were deallocated. Although this covers most properties to be verified (i.e., detecting memory leaks and deallocation of unallocated locations), this is partial semantics because those addresses can still not be returned from an allocation.

Sorts and operations ranging over sequences of addresses Program arrays can be modelled by introducing an additional $\mathit{AddrRange}$ sort and related operations, such as those shown in Table 3. batchAlloc would return an $\mathit{AddrRange}$ containing n addresses, where n is its last argument. batchWrite can be used to *batch* update an address range, nthInRange to extract an address from an address range, contains to check if an address range contains an address, and validRange to check whether all addresses in a range are allocated. In our tools TriCera and Eldarica, we are already using this extended version of the theory of heaps.

5. Related Work

Separation Logic extends the assertions of Hoare's logic [20] to succinctly express properties of heap and shared mutable data-structures [21]. Research has been done on specialised decision procedures for separation logic in SMT [22, 23], and there is a proposal for encoding separation logic in SMT-LIB 2.5 [24].

The theory of heaps and separation logic both provide mechanisms for reasoning about the heap; however, their approaches are orthogonal. Separation logic extends the assertion

language with additional operators, while the theory of heaps provides an interchange format for encoding programs with the goal of preserving as much information about the heap as possible. Both could be used in a complementary way to encode program assertions and the program itself.

Linear Maps provide a similar proof strategy to that of separation logic, while staying within the confines of classical logic [25]. The authors describe a two-way *erasure transformation*, transforming between imperative programs with a single unified heap and programs with multiple disjoint linear maps. Since the transformation is completely in classical logic, off-the-shelf SMT solvers and theorem provers can be used without a special decision procedure by making use of the existing theories such as the theory of arrays and the theory of sets.

Unlike the transformational approach of linear maps, the theory of heaps aims to defer the handling of heap to the solvers. In fact, the linear maps strategy could also make use of the theory of heaps in order to have access to more specialised decision procedures, and not be restricted to the theory of arrays.

Other related work The authors of [26] extend an SMT solver with a decision procedure to decide unbounded heap reachability with support for Boolean and integer data fields. [27] also describes a decision procedure for verification of heap-manipulating programs. Both papers are about verifying heap reachability, and both of them highlight the need for a standard theory of heaps as that would have provided a framework for the research and ease the adoption of proposed decision procedures by different solvers.

6. Experiments

In order to highlight the feasibility of using the theory in a more concrete setting, we have collected C benchmarks from the *ReachSafety* and *MemSafety* categories of SV-COMP 2022 [28]. TriCera[1], a CHC-based model checker for C programs, was extended to produce CHCs in the theory of heaps. To create a preliminary set of CHC benchmarks modulo heaps, we filtered out programs that require heap, but none of the features not yet supported by TriCera (e.g., stack pointers, floats etc.). We have also excluded benchmarks that any of the tested tools reported a parsing error for, in the end, 361 benchmarks remained.

In order to focus on the evaluation of the theory of heaps rather than that of bit-vectors (which are already challenging for Horn solvers on their own [29]), integer types in the CHCs were encoded using mathematical integers. This meant some benchmarks did not return their expected result, as some of them depend on the correct modelling of overflow; however, no conflicting answers were observed in the results by Eldarica and Z3/Spacer.

We then compared the performance of different solvers on those 361 benchmarks. As tools providing (early) native support for the theory of heaps, the SMT solver Princess [30] was extended to support the theory using the reasoning and interpolation procedures from [17], and the CHC solver Eldarica [31] was extended to make use of the newly added theory in Princess. Other solvers can process CHCs output by TriCera after converting constraints

[1] https://github.com/uuverifiers/tricera

Table 4
Results for the 361 heap benchmarks with Eldarica, Z3/Spacer, and CPAchecker. Eldarica is run on both heap benchmarks and their array encodings. Z3/Spacer is only run on the array encoded benchmarks. *Portfolio* row shows the combined best results of the previous three rows. The last row shows CPAchecker's performance on the same set of benchmarks using their original C files.

	sat	unsat	unknown
Eldarica (heap)	14	40	307
Eldarica (array)	**41**	**57**	263
Z3/Spacer (array)	25	47	289
Portfolio	**42**	63	256
CPAchecker	7	**71**	283

in the theory of heaps to array constraints; we used an extended[2] version of the encoding given in [32] implemented in the tool *heap2array*[3] for this conversion. Both Eldarica and Z3/Spacer [33] were run on the array versions of the benchmarks. Lastly, we provide results that CPAchecker [34] (version 2.1.1), one of the top C model checkers [35], produced on the source SV-Comp benchmarks. We have made available all benchmarks used in our experiments [36].

The experiments were run on an AMD Opteron 2220 SE (2.8 GHZ with 4 CPUs) machine running 64-bit Linux with 6 GB of RAM and a wall-clock timeout of 900 seconds.

The results are shown in Table 4. In the first three rows, Eldarica with the array encoded benchmarks performed best. One benchmark could only be solved by Eldarica (heap), 16 benchmarks only by Eldarica (array) and one benchmark only by Z3/Spacer. The best results that can be obtained using a *portfolio* approach (i.e., running the solvers in parallel and taking the first result) are shown in the *Portfolio* row. The last row shows CPAchecker's performance on the source C programs with the combined results of checking memory safety and reachability properties. There were 27 benchmarks that could only be solved by CPAchecker.

The comparison with CPAchecker shows that real-world C programs can indeed be encoded and analysed using the proposed theory of heaps, resulting in a competitive verification tool. Although our current decision procedure for the theory of heaps does not perform as well as its array counterpart yet, the theory provides a uniform representation of programs that can easily be mapped to different back-ends.

7. Conclusions and Outlook

We have proposed a theory of heaps, along with its syntax and semantics, and discussed possible alternative definitions and extensions. The intention is that the ideas presented here will initiate discussions, and eventually result in a common interchange language for programs with heap.

[2] encoding of sorts and operations ranging over sequences of addresses discussed in Section 4
[3] https://github.com/zafer-esen/heap2array

As a long-term goal, we would like to include a heap track also at the CHC-COMP competition; a part of the benchmarks in the LIA-nonlin-Arrays-nonrecADT category of CHC-COMP 2022 already stems from heap theory benchmarks, using an encoding into the theory of arrays.

The algorithms from [17] are direct and unrefined adaptions of procedures for the theory of arrays, and more work is needed to obtain, e.g., practical interpolation methods. However, now that the design choice is shifted to the solvers, alternative approaches can be employed to improve the results without changing the CHC representation of programs. In this context, two directions we are currently pursuing are improved decision and interpolation procedures for the heap theory, and the adaptation of the invariant-based heap encoding used in JayHorn [4].

References

[1] S. Kalra, S. Goel, M. Dhawan, S. Sharma, ZEUS: analyzing safety of smart contracts, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society, 2018. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf.

[2] S. Grebenshchikov, N. P. Lopes, C. Popeea, A. Rybalchenko, Synthesizing software verifiers from proof rules, in: J. Vitek, H. Lin, F. Tip (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, ACM, 2012, pp. 405–416. URL: https://doi.org/10.1145/2254064.2254112. doi:10.1145/2254064.2254112.

[3] A. Gurfinkel, T. Kahsai, A. Komuravelli, J. A. Navas, The SeaHorn verification framework, in: D. Kroening, C. S. Pasareanu (Eds.), Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, volume 9206 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 343–361. URL: https://doi.org/10.1007/978-3-319-21690-4_20. doi:10.1007/978-3-319-21690-4_20.

[4] T. Kahsai, R. Kersten, P. Rümmer, M. Schäf, Quantified heap invariants for object-oriented programs, in: T. Eiter, D. Sands (Eds.), LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017, volume 46 of *EPiC Series in Computing*, EasyChair, 2017, pp. 368–384. URL: https://easychair.org/publications/paper/Pmh.

[5] Y. Matsushita, T. Tsukada, N. Kobayashi, RustHorn: CHC-based verification for Rust programs, in: P. Müller (Ed.), Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, volume 12075 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 484–514. URL: https://doi.org/10.1007/978-3-030-44914-8_18. doi:10.1007/978-3-030-44914-8_18.

[6] R. Sato, N. Iwayama, N. Kobayashi, Combining higher-order model checking with refinement type inference, in: M. V. Hermenegildo, A. Igarashi (Eds.), Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019, ACM, 2019, pp. 47–53. URL: https://doi.org/10.1145/3294032.3294081. doi:10.1145/3294032.3294081.

[7] P. Rümmer, Competition report: CHC-COMP-20, in: L. Fribourg, M. Heizmann (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, volume 320 of *EPTCS*, 2020, pp. 197–219. URL: https://doi.org/10.4204/EPTCS.320.15. doi:10.4204/EPTCS.320.15.

[8] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Technical Report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[9] A. Komuravelli, N. Bjørner, A. Gurfinkel, K. L. McMillan, Compositional verification of procedural programs using Horn clauses over integers and arrays, in: R. Kaivola, T. Wahl

(Eds.), Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015, IEEE, 2015, pp. 89–96.

[10] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Program verification using constraint handling rules and array constraint generalizations, Fundam. Inform. 150 (2017) 73–117. URL: https://doi.org/10.3233/FI-2017-1461. doi:10.3233/FI-2017-1461.

[11] P. M. Rondon, M. Kawaguchi, R. Jhala, Liquid types, in: R. Gupta, S. P. Amarasinghe (Eds.), Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, ACM, 2008, pp. 159–169. URL: https://doi.org/10.1145/1375581.1375602. doi:10.1145/1375581.1375602.

[12] N. Bjørner, K. L. McMillan, A. Rybalchenko, On solving universally quantified Horn clauses, in: F. Logozzo, M. Fähndrich (Eds.), Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, volume 7935 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 105–125. URL: https://doi.org/10.1007/978-3-642-38856-9_8. doi:10.1007/978-3-642-38856-9_8.

[13] D. Monniaux, L. Gonnord, Cell morphing: From array programs to array-free Horn clauses, in: X. Rival (Ed.), Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, volume 9837 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 361–382. URL: https://doi.org/10.1007/978-3-662-53413-7_18. doi:10.1007/978-3-662-53413-7_18.

[14] Z. Esen, P. Rümmer, A theory of heap for constrained Horn clauses (extended technical report), CoRR abs/2104.04224 (2021). URL: https://arxiv.org/abs/2104.04224. arXiv:2104.04224.

[15] Z. Esen, P. Rümmer, Towards an SMT-LIB theory of heap (extended abstract), in: L. Fribourg, M. Heizmann (Eds.), 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, volume 320 of *EPTCS*, 2020.

[16] Z. Esen, P. Rümmer, Abstract: Towards an SMT-LIB theory of heap, in: F. Bobot, T. Weber (Eds.), Proceedings of the 18th International Workshop on Satisfiability Modulo Theories colocated with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Online (initially located in Paris, France), July 5-6, 2020, volume 2854 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, p. 60. URL: http://ceur-ws.org/Vol-2854/abstract4.pdf.

[17] Z. Esen, P. Rümmer, Reasoning in the theory of heap: Satisfiability and interpolation, in: M. Fernández (Ed.), Logic-Based Program Synthesis and Transformation, LNCS, Springer, Cham, 2021, pp. 173–191.

[18] N. Bjørner, A. Gurfinkel, K. L. McMillan, A. Rybalchenko, Horn clause solvers for program verification, in: L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, W. Schulte (Eds.), Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, volume 9300 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 24–51. URL: https://doi.org/10.1007/978-3-319-23534-9_2. doi:10.1007/978-3-319-23534-9_2.

[19] J. McCarthy, Towards a mathematical science of computation, in: Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962, North-Holland, 1962, pp. 21–28.

[20] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580. URL: https://doi.org/10.1145/363235.363259. doi:10.1145/363235.363259.

[21] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings, IEEE Computer Society, 2002, pp. 55–74. URL: https://doi.org/10.1109/LICS.2002.1029817. doi:10.1109/LICS.2002.1029817.

[22] A. Reynolds, R. Iosif, C. Serban, T. King, A decision procedure for separation logic in SMT, in: C. Artho, A. Legay, D. Peled (Eds.), Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings, volume 9938 of *Lecture Notes in Computer Science*, 2016, pp. 244–261. URL: https://doi.org/10.1007/978-3-319-46520-3_16. doi:10.1007/978-3-319-46520-3_16.

[23] J. A. N. Pérez, A. Rybalchenko, Separation logic modulo theories, in: C. Shan (Ed.), Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings, volume 8301 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 90–106. URL: https://doi.org/10.1007/978-3-319-03542-0_7. doi:10.1007/978-3-319-03542-0_7.

[24] R. Iosif, C. Serban, A. Reynolds, M. Sighireanu, Encoding separation logic in smt-lib v2.5, 2018. URL: https://sl-comp.github.io/docs/smtlib-sl.pdf.

[25] S. K. Lahiri, S. Qadeer, D. Walker, Linear maps, in: R. Jhala, W. Swierstra (Eds.), Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011, ACM, 2011, pp. 3–14. URL: https://doi.org/10.1145/1929529.1929531. doi:10.1145/1929529.1929531.

[26] Z. Rakamaric, R. Bruttomesso, A. J. Hu, A. Cimatti, Verifying heap-manipulating programs in an SMT framework, in: K. S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (Eds.), Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings, volume 4762 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 237–252. URL: https://doi.org/10.1007/978-3-540-75596-8_18. doi:10.1007/978-3-540-75596-8_18.

[27] S. Lahiri, S. Qadeer, A decision procedure for well-founded reachability, 2007.

[28] D. Beyer, SV-Benchmarks: Benchmark Set for Software Verification and Testing (SV-COMP 2022 and Test- Comp 2022), 2022. URL: https://doi.org/10.5281/zenodo.5831003. doi:10.5281/zenodo.5831003.

[29] P. Backeman, P. Rümmer, A. Zeljic, Bit-vector interpolation and quantifier elimination by lazy reduction, in: N. S. Bjørner, A. Gurfinkel (Eds.), 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE, 2018, pp. 1–10. URL: https://doi.org/10.23919/FMCAD.2018.8603023. doi:10.23919/FMCAD.2018.8603023.

[30] P. Rümmer, 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, volume 5330 of *LNCS*, Springer, 2008, pp. 274–289.

[31] H. Hojjat, P. Rümmer, The ELDARICA horn solver, in: N. Bjørner, A. Gurfinkel (Eds.), 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, IEEE, 2018, pp. 1–7. URL: https://doi.org/10.23919/FMCAD.2018.8603013. doi:10.23919/FMCAD.2018.8603013.

[32] Z. Esen, P. Rümmer, A theory of heap for constrained Horn clauses (extended technical report), CoRR abs/2104.04224 (2021). URL: https://arxiv.org/abs/2104.04224. arXiv:2104.04224.

[33] A. Komuravelli, A. Gurfinkel, S. Chaki, E. M. Clarke, Automatic abstraction in SMT-based unbounded software model checking, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 846–862. URL: https://doi.org/10.1007/978-3-642-39799-8_59. doi:10.1007/978-3-642-39799-8_59.

[34] D. Beyer, M. E. Keremoglu, CPAchecker: A tool for configurable software verification, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 184–190. URL: https://doi.org/10.1007/978-3-642-22110-1_16. doi:10.1007/978-3-642-22110-1_16.

[35] D. Beyer, Progress on software verification: SV-COMP 2022, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II, volume 13244 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 375–402. URL: https://doi.org/10.1007/978-3-030-99527-0_20. doi:10.1007/978-3-030-99527-0_20.

[36] Z. Esen, P. Rümmer, TriCera Benchmarks: SMT-LIB Encodings of SV-COMP 2022 Benchmarks by TriCera, 2022. URL: https://doi.org/10.5281/zenodo.6950363. doi:10.5281/zenodo.6950363.