Towards an SMT-LIB Theory of Heap

Zafer Esen  Philipp Rümmer
Uppsala University, Sweden

1 Introduction

Constrained Horn Clauses (CHC) are a convenient intermediate verification language that can be generated by several verification tools, and that can be processed by several mature and efficient Horn solvers. One of the main challenges when using CHC in verification is the encoding of program with heap-allocated data-structures: such data-structures are today either represented explicitly using the theory of arrays (e.g., [4, 2]), or are transformed away with the help of invariants or refinement types (e.g., [6, 1, 5, 3]). Both approaches have disadvantages: they are low-level, do not preserve the structure of a program well, and leave little design choice with respect to the handling of heap to the Horn solver. This abstract presents ongoing work on the definition of a high-level SMT-LIB theory of heap, which in the context of CHC gives rise to standard interchange format for programs with heap data-structures. The abstract presents the signature and intuition behind the theory. A preliminary version of the theory axioms can be found in the appendix. The abstract is meant as a starting point for discussion, and request for comments.

A theory of heap has to cover a number of functionalities, including: (i) representation of the type system associated with heap data, and of pointers; (ii) reading and updating of data on the heap; (iii) handling of object allocation. In our proposal, we use algebraic data-types (ADTs), as already standardised by the SMT-LIB, as a flexible way to handle (i); for (ii) our theory offers operations akin to the theory of arrays, and (iii) is provided by additional allocation functions. The theory is deliberately kept simple, so that it is easy to add support to Horn solvers: a Horn solver can, for instance, internally encode heap using the existing theory of arrays, or implement transformational approaches like [1, 5]. 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. (Object-local pointer arithmetic could be handled in a verification system before encoding a program as CHCs.)

2 Syntax and Semantics of the Theory

In order to explain the syntax and semantics of the theory, we start with a simple example program. The example defines a pair named Node in a C-like language, and then creates a linked list on the heap using this type. The new syntax is shorthand for a memory allocation function, which allocates memory on the heap using the passed object, which in this case is a Node.

```
1 struct Node { int data; Node* next; };
2 Node* list = new Node(0, NULL);
3 list->next = new Node(list->data + 1, NULL);
```

To encode the program using our theory, first a heap has to be declared that covers the program types, as shown in the upper half of Listing 1. Each heap declaration introduces several sorts: a sort Heap of heaps; a sort Address of heap addresses, and a number of data-type used to represent heap data. The

---

1 Algebraic Data Types (ADTs)
Listing 1: An SMT-LIB declaration of a Heap and CHC for the example program given in Prolog syntax. Note that all variables are implicitly universally quantified with the correct sort (e.g., \( \forall h : \text{Heap} \)). The clauses are given in Prolog notation for the sake of brevity, which could also be written in SMT-LIB.

```
(declare-sort Heap (name of the heap sort to declare ))
(declare-sort Address (name of the Address sort to declare ))
(declare-sort Object (object sort, usually one of the data-types ))
((Object 0) (Node 0)) ; the default object stored at unallocated addresses
(((WrappedInt 0) ) ) ; data-types
(((Wrapped (getInt Int)) ) ) ; constructors for sort Object
(((WrappedNode (getNode Node)) ) ) ; constructors for sort Node
(((Node (data Int) (next Address))) ) ; constructors for sort Node

: 11 ; CHC below are given in Prolog notation instead of SMT-LIB syntax for brevity.
11 (emptyHeap).
12 (emptyHeap, ar._1, ar._2) :- I1(h), ar = allocate(h, WrappedNode(Node(0, NULL))).
13 (h, list, n) :- I2(h, list), n = getNode(read(h, list)).
14 false :- I2(h, list), !valid(h, list).
15 false :- I2(h, list), !isWrappedNode(read(h, list)).
16 false :- I3(h, list, n, ar._2) :- I3(h, list, n),
17 ar = allocate(h, WrappedNode(Node(data(n)+1, NULL))).
18 false :- I4(h, list, n, p),
19 !isWrappedNode(read(h, list)).
20 h = write(h, list, WrappedNode(Node(x, p)));
21 false :- I4(h, list, n, p), !valid(h, list).
22 false :- I4(h, list, n, p), !isWrappedNode(read(h, list)).
```

data-type declaration is integrated into the heap declaration because heap objects naturally have to store heap addresses, but is otherwise equivalent to a declare-datatypes command in SMT-LIB 2.6.

Data-types are used to represent the hierarchy of types on the heap, and allow us to have just a single sort for all objects on the heap. Consider the example program, in which type \( \text{Int} \) maps to the mathematical integers \( \mathbb{Z} \), and \( \text{Node} \) type using a data-type \( \text{Node} \) with two fields \( \text{data} \) and \( \text{next} \). We can assume that the only types stored on the heap are \( \text{Int} \), \( \text{Node} \), and \( \text{Address} \). Again using data-types, these sorts can be \emph{wrapped} to be represented as a single \( \text{Object} \) sort. In the declaration of the \( \text{Heap} \) in Listing 11 the \( \text{Object} \) sort is defined as a data-type with wrappers for each possible sort on the heap (lines 7–9), and \( \text{Node} \) as a record with a single constructor (line 10). To get back the data stored in an object, a single selector is defined for each wrapped sort, which is named a \emph{getter}.

Data-types are a clean and flexible way to represent types in programs. In C, note that nested structs, enums, and unions can all easily be mapped to data-types, while recursive data-types could be used for strings or arrays (although it is probably more efficient to natively integrate the theory of arrays for this purpose). Inheritance in Java-like languages can be represented through a \emph{parent} field added to subclasses, and multiple-inheritance in C++ through multiple \emph{parent} fields; sub-typing becomes explicit.

### 2.1 Encoding in Horn Clauses

The example program can be encoded in Horn clauses using Prolog notation as given in Listing 11 and the used operations are introduced in Section 2.2.

Line 12 creates an empty heap term, and in line 13, memory is allocated on the heap using a zero-initialised Node Object. Allocation returns a pair: the new heap after allocation (ar._1), and the address of the allocated location (ar._2), which in SMT-LIB can be handled through a further data-type AllocationResult. The returned Address value ar._2 is assigned to list.
nullAddress : () \rightarrow Address
emptyHeap : () \rightarrow Heap
allocate : Heap \times Object \rightarrow Heap \times Address (AllocationResult)
read : Heap \times Address \rightarrow Object
write : Heap \times Address \times Object \rightarrow Heap
valid : Heap \times Address \rightarrow Bool

Table 1: Functions and predicates of the theory

On line 14, list is read from the heap and the returned Node is stored in the temporary variable n, and lines 15–16 assert that this heap access is safe. In C, such checks are relevant for each read and write, due to the possibility of pointer casts and of uninitialised pointers, whereas in Java corresponding checks should be added for reference typecasts. Line 15 can be read as: “It is not the case that I2 holds and Heap h at Address list is unallocated.” Line 16 ensures that the read Object is actually of type Node, using the tester available for each data-type constructor. Lines 22–23 contain similar checks for the other heap accesses in the program.

The clause at lines 17–18 allocates memory for the second Node, and assigns the returned Address to the temporary variable p (i.e., the last argument of I4). Lines 19–21 update the list’s next field, by creating a new Node where the data field remains the same, and the next field is assigned the value of p. The constructed Node is then wrapped and written to the the Address pointed to by list.

Representing all pointer types using the single sort Address simplifies the theory; however, an Address has no associated type information. This is closer in semantics to languages like C, where casts between arbitrary pointer types are possible; in languages with a stronger type system, like Java, some memory safety assertions (e.g., lines 16 and 23 in Listing 1) can be turned into assumptions, since those properties are guaranteed by the type system.

2.2 Functions and Predicates of the Theory

Functions and predicates of the theory are given in Table 1. Function nullAddress returns an Address which is always unallocated/invalid, while emptyHeap returns the Heap that is unallocated everywhere.

Function allocate takes a Heap and an Object, and returns an AllocationResult. AllocationResult is a data-type representing the pair \( (Heap, Address) \). The returned Heap at Address contains the passed Object, with all other locations unchanged.

Functions read and write are similar to the array select and store operations; however, unlike an array, a heap is a partial mapping from addresses to objects. This means the read and write functions only behave as their array counterparts if the heap is allocated at the address being read or written to.

The behavior of accessing unallocated memory locations is undefined in many languages. Regarding reads, we left the choice to the user of the theory, who can designate a default Object in the heap declaration (line 5 in Listing 1). This Object, which we named defObj in our axioms, is returned on an invalid read. The function write normally returns a new Heap if the address that is being accessed was allocated. If not, then the original Heap is returned without any changes. (An alternative semantics would be to return the emptyHeap; however, returning the same heap simplifies the read-over-write axiom by removing the validity check from the left-hand side of the implication). Validity of a write can be checked via memory-safety assertions as shown in Listing 1.

We propose a further short-hand notation nthAddressi, which is not listed in Table 1 but is useful when presenting satisfying assignments. It is used to concisely represent Address values which would be returned after i allocate calls. This becomes possible with a deterministic allocation axiom added to the theory.
References


Appendix A  Axioms of the Theory

Axioms of the theory are given in Table 2. Note that all variables in the table are universally quantified with sorts $h : \text{Heap}$, $p : \text{Address}$, $o : \text{Object}$ and $\text{ar} : \text{AllocationResult}$. Variables can also appear subscripted. Let $\text{ar}$ be an $\text{AllocationResult}$, which is the pair $\langle \text{Heap}, \text{Address} \rangle$. We use the notation $\text{ar}_1$ and $\text{ar}_2$ to select the Heap and Address fields of $\text{ar}$, respectively.

As explained in Section 2.2, $\text{defObj}$ is a user-specified term returned on invalid reads.

| valid($h, p$) $\implies$ read(write($h, p, o$), $p$) = $o$ | read-over-write | [row1] |
| $p_1 \neq p_2$ $\implies$ read(write($h, p_1, o$), $p_2$) = read($h, p_2$) | read-over-write | [row2] |
| allocate($h, o$) = $\text{ar} \implies$ read($\text{ar}_1, \text{ar}_2$) = $o$ | read-over-allocate | [roa1] |
| allocate($h, o$) = $\text{ar} \land p \neq \text{ar}_2 \implies$ read($\text{ar}_1, p$) = read($h, p$) | read-over-allocate | [roa2] |
| $\neg$valid($h, p$) $\implies$ write($h, p, o$) = $h$ | invalid write | [ivwt] |
| $\neg$valid($h, p$) $\implies$ read($h, p$) = $\text{defObj}$ | invalid read | [ivrd] |
| $\neg$valid(\text{emptyHeap}, $p$) | empty heap validity | [vld1] |
| $\neg$valid($h, \text{nullAddress}$) | $\text{nullAddress}$ validity | [vld2] |

$\forall p, \text{ar}_2 \neq p \implies (\text{valid($h, p$)} \iff \text{valid($\text{ar}_1, \text{ar}_2$)})$  [alloc1]

$\forall p. (\text{valid($h_1, p$)} \iff \text{valid($h_2, p$)} \land \text{read($h_1, p$)} = \text{read($h_2, p$)})$  [ext]

$\forall p. \text{valid($h_1, p$)} \iff \text{valid($h_2, p$)} \implies h_1 = h_2$  [alloc2]

$\exists f : \text{Nat} \rightarrow \text{Heap}, g : \text{Nat} \rightarrow \text{Address}.$

$\forall i : \text{Nat}. (f(i + 1), g(i + 1)) = \text{allocate}(f(i), \text{defObj}) \land$

$\forall p : \text{Addr}. \exists i : \text{Nat}. g(i) = p$  [cons]

Table 2: Axioms of the theory

[row1] Reads from an Address $p$ that was allocated and previously written to using Object $o$, returns $o$. This is similar to the array read-over-write axiom; however, here it is only applied when the accessed location was previously allocated.

[row2] Reads from Address $p_2$ from a Heap $h$ written at Address $p_1$ is the same as directly reading Address $p_2$ from Heap $h$. This axiom is exactly the same as the corresponding array read-over-write axiom. Checking for validity here is not required due to the axiom [ivwt].

[roa1] Reading from a Heap, using the Address returned from an allocation using Heap $h$ and Object $o$, returns $o$.

[roa2] Reading from a Heap using an Address $p$ that is different than the Address returned from the allocation, which was done using Heap $h$ and Object $o$, is the same as directly reading $p$ from $h$.

[ivwt] A write to an invalid Address of Heap $h$ returns $h$.

[ivrd] A read from an invalid Address of a Heap returns the defObj term.

[vld1] Empty heap is completely unallocated.

[vld2] nullAddress is unallocated in any Heap.
Towards an SMT-LIB Theory of Heap

[alloc1] Allocation takes a Heap $h$, and returns a new Heap where the Address is allocated (i.e. valid). It also says that the newly allocated Address must have been unallocated at $h$. The last conjunct in the axiom states that the validity of both Heaps are the same at all Addresses except the one which was just allocated.

[alloc2] This axiom is to ensure that the allocations are done in a deterministic fashion. If two Heaps were both allocated at exactly the same Addresses (i.e. a result of the same number of allocate calls), then allocating a new Object on any of these two Heaps will return the same Address.

[ext] The extensionality axiom states that, given any Address $p$, if both Heaps are allocated at $p$, and reads from $p$ return the same Object in both of them, then the two Heaps must be the same.

[cons] This axiom makes the Heap constructable by enumerating every Heap and Address. This can also be expressed as an induction axiom. defObj in the axiom represents an arbitrary term.
Appendix B  Complete SMT-LIB Encoding of the Example Program

```plaintext
; Complete SMT-LIB encoding of the example program
(declare-heap Heap ; name of the heap sort to declare
Address ; name of the Address sort to declare
Object ; object sort, usually one of the data-types
(WrappedInt 0) ; the default object stored at unallocated addresses
((Object 0) (Node 0)) ; data-types
(((WrappedInt (getInt Int)) ; constructors for sort Object
(WrappedNode (getNode Node)))
(((WrappedAddr (getAddr Address)))
((Node (data Int) (next Address)))))) ; constructors for sort Node

; Object, Node, AllocationResult are declared as side-effect of the heap declaration:
(declare-datatypes ((Object 0) (Node 0) (AllocationResult 0))
(((WrappedInt (getInt Int))
(WrappedNode (getNode Node))
(WrappedAddr (getAddr Address)))
((Node (data Int) (next Address))))

(decl-fun I1 (Heap Bool) ; <h>
(decl-fun I2 (Heap Address) Bool) ; <h, list>
(decl-fun I3 (Heap Address Node) Bool) ; <h, list, n>
(decl-fun I4 (Heap Address Node Address) Bool) ; <h, list, n, p>
(decl-fun I5 (Heap Address) Bool) ; <h, list>

; I1 (emptyHeap).
(assert (I1 emptyHeap))

; I2 (ar.1, ar.2) := I1(h), ar = allocate(h, WrappedNode(Node(0, NULL))).
(assert (forall ((h Heap) (list Address)) (ar AllocationResult))
  =>
  (and (I1 h) (= ar (allocate h (WrappedNode (Node 0 NULL))))
       (I2 _1 ar _2 ar)))

; I3(h, list, n) := I2(h, list), n = getNode(read(h, list)).
(assert (forall ((h Heap) (list Address) (n Node))
  =>
  (and (I2 h list) (= n (getNode (read h list)))
       (I3 h list n))))

; false := I2(h, list), !valid(h, list).
(assert (forall ((h Heap) (list Address))
  =>
  (and (I2 h list) (not (valid h list))
       false))

; false := I2(h, list), !isWrappedNode(read(h, list)).
(assert (forall ((h Heap) (list Address))
  =>
  (and (I2 h list) (not (isWrappedNode (read h list)))
       false))

; I4(ar.1, list, n, ar.2) := I3(h, list, n),
  ar = allocate(h, WrappedNode(Node(data(n)+1, NULL))).
(assert (forall ((h Heap) (list Address) (ar AllocationResult) (n Node))
  =>
  (and (I3 h list n)
       (= ar (allocate h (WrappedNode (Node (+ (data n) 1) NULL))))
       (I4 _1 ar list n _2 ar))))
```

Towards an SMT-LIB Theory of Heap

61  \(I_5(h, \text{list}) : \neg I_4(h, \text{list}, n, p).\)
62  ;
63  \(x = \text{data}(\text{getNode}(\text{read}(h, \text{list}))).\)
64  \(h_1 = \text{write}(h, \text{list}, \text{WrappedNode}(\text{Node}(x, p))).\)
65  (assert (forall ((h Heap) (h_1 Heap) (\text{list} Address) (n Node) (p Address) (x Int))
66    (=>
67      (and (I4 h \text{list} n p)
68         (= x (\text{data}(\text{getNode}(\text{read} h \text{list}))))
69         (= h_1 (\text{write} h \text{list} (\text{WrappedNode}(\text{Node} x p))))))
70      (I5 h_1 \text{list}))))
71  ; false :- I4(h, \text{list}, n, p), \neg \text{valid}(h, \text{list}).
72  (assert (forall ((h Heap) (\text{list} Address) (n Node) (p Address))
73    (=>
74      (and (I4 h \text{list} n p) (\neg \text{valid} h \text{list})))
75      false))
76  ; false :- I4(h, \text{list}, n, p), \neg \text{isWrappedNode}(\text{read}(h, \text{list})).
77  (assert (forall ((h Heap) (\text{list} Address) (n Node) (p Address))
78    (=>
79      (and (I4 h \text{list} n p) (\neg \text{isWrappedNode}(\text{read} h \text{list}))))
80      false))