A Theory of Heaps for Constrained Horn Clauses

Zafer Esen and Philipp Rümmer

Uppsala University, Sweden

Abstract

Constrained Horn Clauses (CHCs) are an intermediate program representation 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 abstract presents ongoing work on an *SMT-LIB theory of heaps* tailored to CHCs, with the goal of enabling a standard interchange format for programs with heap data-structures.

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 [9] to verification of computer programs in various languages [5, 6, 8]. One of the main challenges when using Constrained Horn Clauses (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., [10, 2]). This is a natural encoding because a heap can be seen as an array of memory locations; 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., [12, 1, 11, 8]). 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). This is because every heap access is replaced with assertions and assumptions about local object invariants, so that global program invariants might not be expressible.

Both approaches leave little design choice with respect to handling of heaps to CHC solvers. Dealing with heaps at encoding level implies repeated effort when designing verifiers for different programming languages, makes it hard to compare different approaches to encode heaps, 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 heaps.

This abstract presents ongoing work that aims to extend CHCs to a standardised interchange format for programs with heap data-structures. We briefly present the theory, which does not restrict the way in which CHC solvers approach heaps, while covering the main functionality of heaps 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 heaps using the existing theory of arrays, or implement transformational approaches like [1, 11].

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 }
```

Being language-agnostic, the theory allows for common encodings across different applications, and is in the spirit of both CHCs and SMT-LIB. We refer the reader to [4] for a more comprehensive look at the theory.

2 The Theory of Heaps

Listing 1 shows a simple Java program which constructs a singly-linked list, highlighting various heap interactions such as allocating objects on the heap (lines 26–27), as well as reading (lines 28–29) and modifying (line 28) heap data.

To encode this program using the theory of heaps, first a heap has to be declared that covers the program types. Each heap comes with its own sorts for the heap itself (*Heap*) and for heap addresses (*Address*). Within one heap, all pointers are represented using its single *Address* sort.

The function emptyHeap is then used to instantiate an empty heap, and allocations are done by using the allocate function of the theory. read and write functions are used to read from and write to heap locations, respectively. The predicate valid allows checking whether an accessed location is valid. valid, along with the tester methods provided by ADTs, can be used to assert memory and type safety of heap accesses.

The complete SMT-LIB encoding for this program[1] that uses the theory of heaps is given in Appendix A. Below we provide an overview of the sorts and operations of the theory. For a detailed explanation of each operation and its semantics (through axioms and an equivalent array encoding), we refer to the extended technical report on the theory of heaps [4].

Declaration For the program shown in Listing 1, an example declaration of the theory of heaps (in SMT-LIB v2.6 style) is as follows:

```
(declare-heap Heap Addr                          ; declared Heap and Address sorts
 Object O_Empty                                  ; chosen Object sort and 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))))
```

Sorts Each heap declaration introduces several sorts: (i) a sort *Heap* of heaps, (ii) a sort *Address* of heap addresses, (iii) zero or more ADT sorts, one of which is selected during decla-

[1]Try it in Eldarica: http://logicrunch.it.uu.se:4096/~wv/eldarica/?ex=perma%2F1633892407_12575427

ration as the *Object* sort that represents heap data, (iv) an additional ADT sort that holds the pair $\langle Heap, Address\rangle$ which is the result of calling allocate.

Operations

nullAddress	:	()	$\rightarrow$	*Address*
emptyHeap	:	()	$\rightarrow$	*Heap*
allocate	:	$Heap \times Object$	$\rightarrow$	$Heap \times Address$
read	:	$Heap \times Address$	$\rightarrow$	*Object*
write	:	$Heap \times Address \times Object$	$\rightarrow$	*Heap*
valid	:	$Heap \times Address$	$\rightarrow$	*Bool*

On invalid reads (i.e., where valid returns *false* for a given $\langle Heap, Address\rangle$), a *default object* is returned, which is specified while declaring the theory. On invalid writes, the written heap is returned unchanged. Allocation results in a new heap that contains the provided object at the returned address.

3 Conclusions and Future Work

We have briefly discussed the proposed theory of heaps. The intention is that the ideas presented here will initiate discussions, and eventually result in a common interchange language for programs with heaps. As a long-term goal, we would like to include a heap track also at the CHC-COMP competition [14].

There is also ongoing work to extend the theory of heaps with further operations and sorts such as batch allocate and *AddressRange*, which are useful when encoding and accessing arrays on the heap. An initial version of the extensions are used in the C model checker TriCera[2] to encode C arrays.

Part of the ongoing work involves developing a decision procedure for the theory of heaps [3]. For this purpose we have implemented procedures in the Princess SMT solver [13] and in the Eldarica CHC solver [7]. The algorithms used are currently direct and unrefined adaptions of procedures for the theory of arrays, and more work is needed to obtain, e.g., practical interpolation methods.

References

[1] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS 2013*, volume 7935 of *LNCS*, pages 105–125, 2013.

[2] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification using constraint handling rules and array constraint generalizations. *Fundam. Inform.*, 150(1):73–117, 2017.

[3] Z. Esen and P. Rümmer. Reasoning in the theory of heap: Satisfiability and interpolation. In *Logic-Based Program Synthesis and Transformation*, LNCS, pages 173–191, 2021.

[4] Z. Esen and P. Rümmer. A Theory of Heap for Constrained Horn Clauses (Extended Technical Report). *arXiv:2104.04224 [cs]*, Apr. 2021.

[5] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI '12*, pages 405–416, 2012.

[6] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *CAV 2015*, volume 9206 of *LNCS*, pages 343–361, 2015.

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

[7] H. Hojjat and P. Rümmer. The ELDARICA horn solver. In *FMCAD 2018*, pages 1–7, 2018.

[8] T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR-21, 2017*, volume 46 of *EPiC Series in Computing*, pages 368–384, 2017.

[9] S. Kalra, S. Goel, M. Dhawan, and 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.

[10] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan. Compositional verification of procedural programs using Horn clauses over integers and arrays. In *FMCAD 2015*, 2015.

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

[12] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 2008*. ACM, 2008.

[13] P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings, LPAR-15*, volume 5330 of *LNCS*, pages 274–289, 2008.

[14] P. Rümmer. Competition report: CHC-COMP-20. In L. Fribourg and M. Heizmann, editors, *VPT/HCVS@ETAPS 2020*, volume 320 of *EPTCS*, pages 197–219, 2020.

A SMT-LIB Encoding of the Example Program

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 are 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)))
```