Finding Universally Quantified Heap Invariants by Horn Clause Transformations

Zafer Esen[1][0000-0002-1522-6673], Philipp Rümmer[1,2][0000-0002-2733-7098], and Tjark Weber[1][0000-0001-8967-6987]

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

Abstract. A common approach in software verification is to encode a program as a set of Constrained Horn Clauses (CHCs), which are then processed and solved automatically by a CHC solver. To streamline this verification approach for the case of programs operating on mutable linked data-structures, we have in earlier work proposed a *theory of heaps,* defined within the SMT-LIB framework, which enables us to represent programs as CHCs with minimal loss of structural information. By preserving high-level program information in the encoding, the theory of heaps enables CHC solvers to apply various internal techniques for handling program heap; among others, to encode the heap further using the theory of arrays, to apply shape analysis, or to translate to a heap-less program with the help of invariants. This paper explores the third option, developing transformation rules that rewrite a set of CHCs into an equisatisfiable set of CHCs with additional predicates representing heap invariants. The proposed method generalises the notion of *space invariants*, which were previously introduced for verifying Java programs, by lifting the entire transformation process to the CHC level. The paper defines the transformation rules, provides detailed correctness proofs, and discusses the strengths and limitations of the approach. We also outline possible extensions of the method.

1 Introduction

In the context of automatic program verification, one of the key challenges is the automatic discovery of program invariants. Constrained Horn Clauses (CHCs) serve as an intermediate verification language, where program invariants do not need to be explicitly specified. Instead, CHC solvers such as Eldarica [19], Golem [8] and Z3-Spacer [25] can attempt to infer these invariants automatically. Many verification frameworks represent programs as CHCs for this purpose, with front-ends across languages such as C [12,18,15], Java [24] and Rust [28]. Beyond program verification, CHCs are employed in a variety of applications, ranging from the verification of smart contracts [3,11] to synthesis of specifications [34] and programs [16]. The CHC language facilitates a separation of concerns, where verification system designers can focus on high-level strategies for encoding their problems into CHCs, while CHC solver developers

can focus on designing and optimising decision procedures. High-level theories – such as algebraic data-types (ADTs), bit-vectors, and the theory of arrays [29] – which CHC solvers natively support, enable this separation of concerns. These theories provide sorts, functions, and predicates to easily represent and operate over program types.

A key challenge in software verification is the handling of programs operating on the heap, i.e., programs that represent data using mutable linked data-structures [9]. In the context of CHCs, to handle the program heap, verification tools typically pick a specific heap encoding that is applied prior to translating the program to CHCs; this encoding could represent the program heap, for instance, using the theory of arrays [29], using invariants [23], or using prophecy variables [28]. However, this early elimination of the program heap in the verification process runs counter to the separation of concerns that motivates the use of CHCs in the first place, and has significant disadvantages: it leads to highly complex analysis methods (e.g., alias and shape analysis) being implemented again and again in different verification tools, while limiting CHC solvers to solving the low-level CHCs that are obtained after eliminating the heap. To address this issue, in previous work we have presented an SMT-LIB *theory of heaps* [14] that enables the translation of programs to CHCs while keeping the relevant heap operations intact. The theory of heaps makes it possible to preserve high-level heap-related information in the encoding itself, and thus enables CHC solvers to use more high-level decision procedures and transformations for handling heap at the CHC level. In this paper we present one such transformation.

Our approach is based on the concept of *space invariants* [23], first defined in the context of the CHC-based verification tool JayHorn [24]. The approach is a program transformation that abstracts heap interactions with *assert* and *assume* statements over symbolic invariants that summarise the possible states of objects on the heap. The transformed program does not contain any heap interactions, therefore leads to a CHC representation that is free of explicit heap operations. This program transformation is *sound* (i.e., the transformed program is safe only if the original program is safe – "safe" meaning the program does not violate a functional or memory-safety property), but it is *incomplete* (i.e., the transformed program can be unsafe even if the original program is safe).

Our method, which we call *heap invariants*, generalises the concept of space invariants by lifting it from Java programs to CHCs, where the heap is represented using the theory of heaps. This generalisation makes the approach independent of the programming language and implementable at the level of CHC solvers. Moreover, in contrast to the space invariant transformation, our transformation is *complete*. Finally, the space invariants approach in [23] is sound only when there are no uninitialised memory accesses. Our approach does not rely on this assumption.

The space invariant method has already been evaluated experimentally using the implementation in JayHorn, including various refinements of the method. In this paper, we therefore focus on the more theoretical question of how the space invariant encoding can be lifted to the level of CHCs, and how the transformation

```
1  typedef struct Node {
2    int data;
3    struct Node* next;
4  } Node;
5
6  void main() {
7    Node* a = NULL;
8    while (nondetInt()) {
9      Node* t = new Node(3, a);
10     assert(I(t, Node(3, a)));
11     a = t;
12   }
13   while (a) {
14     Node n = *a;
15     assume(I(a, n));
16     assert(n.data == 3);
17     a = n.next;
18   }
19 }
```

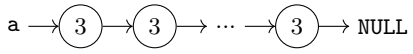

Fig. 1: Left: A C-like program that allocates and iterates over a linked list. The lines 10 and 15 are added by our transformation at the program level using the heap invariant "I". Note that the program-level transformation is shown here for clarity, but our actual transformation operates at the level of CHCs as shown in Figure 2. Our transformation also introduces additional memory-safety constraints, making it applicable even in programs with uninitialised memory accesses. Right: A depiction of the list at the exit of the first loop.

can be shown correct. We provide formal proofs of correctness for both the soundness and completeness of our transformation.

Contributions This paper contributes (i) a sound and complete generalisation of the space invariants transformation at the level of CHCs. This transformation facilitates the inference of universally quantified heap invariants, without assuming the absence of uninitialised memory accesses; and (ii) proofs of correctness.

2 Motivating example

We first illustrate the concept of heap invariants at the level of programs. The C-like program in Figure 1 allocates a linked list in a loop (line 8), writing the value "3" to each node. The list is then traversed in a second loop (line 13), and the property that each node contains the value "3" is asserted at line 16. We ignore the `assume` and `assert` statements at lines 10 and 15 for now. Verifying this program may seem like a simple verification task, but automatic verification turns out to be surprisingly challenging. The size of the list is unbounded, requiring the verification system to infer a quantified invariant that summarises the list's contents. CPAchecker [5,4], a state-of-the-art C program verifier, cannot verify this example. Similarly, CHC-based verifiers SeaHorn [18] and TriCera [15] also fail to verify it. PredatorHP [20,31], a tool based on shape

analysis, can verify the example, but fails when the values written to the list are not constant or cannot be abstractly described using an interval domain.

The verification task can be simplified by summarising the contents of the heap through a *heap invariant,* introduced by the `assume` and `assert` statements at lines 10 and 15. Assume that an oracle (such as a CHC-solver) provided the invariant $I(a, n) \equiv (data(n) = 3)$, capturing the information that the `data` field of all `Node`s ever put on the heap has the value "3". At every update site, we *validate* this invariant by asserting it, as done at line 10 in Figure 1. At each read site, in this case at line 15, we can assume the same invariant. Given the invariant I, verifying the program turns into an easy task that can be carried out automatically by various software model checkers, as the assertion at line 16 is the same as the I provided by our oracle. Without I, the invariant of the loop at line 8 needs to reason about the heap, which is a more complicated universally quantified invariant over all addresses on the heap that is challenging to automatically infer.

The transformation we applied is *sound* and *complete* in general. We formally prove this at the CHC level in Section 6 (Theorem 1 for completeness and Theorem 2 for soundness) and provide the intuition here.

For soundness, we have to prove that, regardless of the chosen I, if the transformed program P' is safe, then also the original program P is safe. Conversely, assume P is unsafe, i.e., the assertion at line 16 fails due to a node with a data value d other than "3". It must then be the case that also P' is unsafe, since the d must have been written to the heap by some heap update operation. Then either d satisfies I, in which case the assertion in line 16 can also fail in P', or otherwise the underlined assertion in line 10 in P' can fail.

We obtain the completeness of the transformation by showing that if P is safe, then there is some I such that P' is also safe. For this, we can simply set I to $\top$, the weakest possible invariant. Then the underlined `assert` and `assume` statements over I have no effect, making P' equivalent to P, and thus P' cannot be unsafe when P is safe.

2.1 Representing the program in CHCs

A Constrained Horn Clause (CHC) is a disjunction of atoms and a constraint over background theories (such as arithmetic, arrays, or heaps), in which each atom typically represents a set of program states. CHCs provide a flexible way to encode the control-flow of programs. A more detailed presentation of CHCs is in Section 4.2.

Figure 2 shows a set of CHCs, where the parts without underlines encode the program in Figure 1 (the underlined parts are added by our transformation). We provide a brief overview of the translating of programs into CHCs, and refer to [7,6], and for CHCs using the theory of heaps to [15,14], for more details.

In Figure 2, the predicates r_1 and r_2 are the loop invariants. The predicate I, added by our transformation, is the heap invariant.

The theory of heaps provides several sorts and operations. The term $h : \mathit{Heap}$ represents the global program heap, and $a : \mathit{Address}$ represents the pointer vari-

$$r_1(h,a) \leftarrow h = \mathsf{emptyHeap} \wedge a = \mathsf{nullAddress} \quad (1)$$

$$r_1(h',a') \leftarrow r_1(h,a) \wedge n = \mathit{Node}(3,a) \wedge (h',a') = \mathsf{allocate}(h,n) \quad (2)$$

$$\underline{I(a',n) \leftarrow r_1(h,a) \wedge n = \mathit{Node}(3,a) \wedge (h',a') = \mathsf{allocate}(h,n)} \quad (3)$$

$$r_2(h,a) \leftarrow r_1(h,a) \quad (4)$$

$$r_2(h,a') \leftarrow r_2(h,a) \wedge n = \mathsf{read}(h,a) \wedge a \neq \mathsf{nullAddress} \wedge a' = \mathit{next}(n) \underline{\wedge \mathsf{valid}(h,a) \wedge I(a,n)} \quad (5)$$

$$\underline{r_2(h,a') \leftarrow r_2(h,a) \wedge n = \mathsf{read}(h,a) \wedge a \neq \mathsf{nullAddress} \wedge a' = \mathit{next}(n) \wedge n = \mathit{defObj} \wedge \neg\mathsf{valid}(h,a)} \quad (6)$$

$$\bot \leftarrow r_2(h,a) \wedge n = \mathsf{read}(h,a) \wedge a \neq \mathsf{nullAddress} \wedge \mathit{data}(n) \neq 3 \underline{\wedge \mathsf{valid}(h,a) \wedge I(a,n)} \quad (7)$$

$$\underline{\bot \leftarrow r_2(h,a) \wedge n = \mathsf{read}(h,a) \wedge a \neq \mathsf{nullAddress} \wedge \mathit{data}(n) \neq 3 \wedge n = \mathit{defObj} \wedge \neg\mathsf{valid}(h,a)} \quad (8)$$

Fig. 2: The CHC representation of the program given in Figure 1 is shown without underlines. The underlined parts are added by our transformation.

able a from the program. read reads from an *Address*, and write updates an *Address*. allocate allocates a new object on the heap and returns its *Address*. emptyHeap returns a *Heap* with no allocated addresses, and nullAddress returns an always-unallocated *Address*. valid checks whether an *Address* is allocated. Lastly *defObj* is the default object returned on invalid reads, defined when declaring the heap theory. A list of heap operations is given in Table 1.

We use mathematical integers for the C `int` type, and the *Address* sort from the theory of heaps for pointers. The C `struct` can be encoded using *algebraic data-types* (ADTs). For the `Node struct`, we define the ADT sort *SNode*. Instances of *SNode* can be constructed using its constructor $\mathit{Node} : \mathit{Integer} \times \mathit{Address} \rightarrow \mathit{SNode}$, with field access via selector functions $\mathit{data} : \mathit{SNode} \rightarrow \mathit{Integer}$ and $\mathit{next} : \mathit{SNode} \rightarrow \mathit{Address}$.

In Figure 2, CHC (1) represents program entry, where the heap is empty, and a is the null address. CHC (2) encodes the body of the first loop, allocating a new *SNode* at each iteration. The transformation adds CHC (3), asserting the predicate I using the the node that was just allocated.

The next CHC (4) from r_1 to r_2 encodes the exit of the first loop. The encoding does not constrain when to exit the first loop, which corresponds to the nondeterministic choice in the guard of the first loop of the program. The remaining CHCs encode the body of the second loop, which iterates as long as $a \neq \mathsf{nullAddress}$.

A heap read can be either valid or invalid: for valid reads, we can assume the heap invariant I holds; for invalid reads the object returned is some error value (defObj) predefined for our program, following the semantics of read. The

additional constraints that use the valid predicate ensure that the transformation remains sound even in the presence of invalid heap accesses.

The CHCs (5) and (7) handle valid reads, and the CHCs (6) and (8) handle invalid reads. The CHCs (5) and (6) capture the continuation of the loop, which advances to the `next` field of the read node in each iteration. Finally, the CHCs (7) and (8) assert that the `data` field of the read node must have the value 3.

Given a set of CHCs, a CHC solver such as Eldarica attempts to find interpretations for the uninterpreted predicates (here r_1, r_2 and I), such that all clauses are satisfiable. If so, the program is considered safe. If any clause is unsatisfiable (i.e., an assertion is violated), the program is unsafe.

The original encoding (without underlines) cannot be verified by Eldarica (currently the only solver supporting the theory of heaps), but the transformed encoding is quickly verified, with I interpreted as $data(n) = 3$.

3 Related Work

First proposed in the context of the CHC-based Java verification tool JayHorn [24], the space invariants approach [23] is inspired by *refinement types* and *liquid types* [33]. The space invariants approach first applies a program transformation to minimise the number of heap reads and writes by merging them into *pull* and *push* operations, where possible. In the second step, push and pull operations are abstracted by introducing symbolic space invariants: a push *asserts* the space invariant, and a pull *assumes* it. The resulting program is free of heap interactions. Unlike our transformation, the original space invariants approach is incomplete, but the authors propose several refinements to improve completeness. Moreover, the space invariants approach requires that there are no uninitialised memory accesses for the transformation to be sound. Our approach is sound even in the presence of uninitialised memory accesses.

Recently, software model checkers have started to support programs that include *uninterpreted predicates* [15,38], which make it possible to encode various different proof rules at the level of programs. This functionality can be used to describe space invariants [38]. Our transformation has similarities to such an encoding, but is applied at the level of CHCs.

A basic decision procedure for the theory of heaps is proposed in earlier work [13]. Although it is both sound and complete, it often fails to infer quantified heap invariants that our transformation facilitates, and fails to solve the example shown in Section 2. Still, our transformation preserves the heap in the CHCs, and remains complete by relying on decision procedures such as the one in [13].

There is a wide range of analysis methods tailored to the verification of heap-manipulating programs: those include shape analysis [35,32,1,10,37,22,17], tree automata [21,2], and type-based techniques [26,30,27]. Our method differs from such tailor-made approaches by relying on a CHC solver as the analysis back-ends, and is as such relatively easy to implement. Our approach is also easy to combine with other kinds of static analysis, in particular methods for value analysis available in CHC solvers. As our transformation preserves the

original heap constraints, our work could be used in combination with many of the mentioned techniques to further enhance heap reasoning.

4 Preliminaries

4.1 First-order logic

Syntax of a many-sorted first-order logic. A *many-sorted signature* in first-order logic is denoted by $\Sigma = (F, P, S, \delta)$, where F is a set of *function symbols*, P is a set of *predicate symbols*, S is a set of *sorts*, and δ is a *sort mapping*. The mapping δ maps each n-ary function to an $(n+1)$-tuple of sorts from S, and each predicate symbol to an n-tuple of sorts. Additionally, we assume a countably infinite set of sorted variables, $\mathcal{X}$, with δ also used to specify the sort of each variable. A variable x with sort s is denoted by $x : s$, where $s = \delta(x)$.

A *term* is a variable or a function symbol applied to its arguments. An *atom* is a predicate symbol applied to its arguments. A *literal* is an atom or its negation. A *clause* is a disjunction of literals. *Formulas* are built from atoms via the usual logical connectives: negation, conjunction, disjunction, implication, as well as existential and universal quantification over sorted variables.

Semantics. A *Σ-structure* is a pair $M = (D, \mathcal{I})$, where D is a sort-indexed family of sets $\{D_s \mid s \in S\}$, and $\mathcal{I}$ is an *interpretation* function that maps each sort $s \in S$ to D_s, each function symbol $f \in F$ with $\delta(f) = (s_1, s_2, \ldots, s_{n+1})$ to a set-theoretic function $\mathcal{I}(f)\colon \mathcal{I}(s_1) \times \mathcal{I}(s_2) \times \cdots \times \mathcal{I}(s_n) \to \mathcal{I}(s_{n+1})$, and each predicate symbol $p \in P$ with $\delta(p) = (s_1, s_2, \ldots, s_n)$ to a relation $\mathcal{I}(p) \subseteq \mathcal{I}(s_1) \times \mathcal{I}(s_2) \times \cdots \times \mathcal{I}(s_n)$. A *variable assignment* β for M maps each variable $x \in \mathcal{X}$ to an element $\beta(x) \in \mathcal{I}(\delta(x))$.

A *theory* T is a pair (Σ_T, M_T) of a many-sorted signature Σ_T and a class of Σ_T-structures M_T. In this paper we assume that the signature contains (at least) the theory of heaps [14], whose operations are interpreted according to their standard semantics (cf. Table 1).

A formula ϕ over some signature Σ is *T-satisfiable* if there exists a Σ_T-structure $M_T = (D_T, \mathcal{I}_T)$ that can be extended to a Σ-structure $M = (D, \mathcal{I})$ (such that $D_T \subseteq D$, and $\mathcal{I}$ restricted to Σ_T coincides with $\mathcal{I}_T$) and a variable assignment β for M, satisfying the formula. We write $\Sigma_T \subseteq \Sigma$ to indicate that Σ is an extension of Σ_T. The symbols in $\Sigma \setminus \Sigma_T$ are called *uninterpreted*, and $\mathcal{I} \setminus \mathcal{I}_T$ is called a *solution* for the uninterpreted symbols.

4.2 Constrained Horn clauses (CHCs)

A Constrained Horn Clause (CHC) in first-order logic is a sentence

$$\forall \bar{x}.\ \big(H \leftarrow (c \wedge \bigwedge_{i=0}^{n} B_i)\big)$$

where H is either an atom or $\bot$, B_i (for $i = 1, \ldots, n$) is an atom, $B_0 = \top$ and c is a constraint over some theories (e.g., the theories of arrays, heaps, etc.). The antecedent of a CHC is called the *body*, and the consequent is called the *head*.

CHCs are typically written using a left arrow for the implication (as above) and by leaving universal quantifiers implicit, which we follow in this paper.

We define R as the set of all non-theory (i.e., uninterpreted) relation symbols, and define $\mathit{Rels}(\mathbb{C})$ as the operator that returns all relation symbols in a set of CHCs $\mathbb{C}$ that are elements of R. When encoding programs with CHCs, these relation symbols can represent sets of program states.

Given a set of CHCs $\mathbb{C}$, a CHC-solver attempts to compute a solution to $\mathit{Rels}(\mathbb{C})$ such that $\mathbb{C}$ is satisfiable. When the CHCs encode the correctness of a program, a solution implies the program is safe, and corresponds to the program invariants. If the CHCs are unsatisfiable, the solver returns a counterexample that corresponds to a program trace from program entry to the violated property.

4.3 Semantics of CHCs

The function $\mathit{val}_{M_T,\beta,\sigma}$ assigns truth values to logical formulas based on the interpretations of symbols and variables. In addition to the structure M_T interpreting the theory symbols, and the variable assignment β, $\mathit{val}_{M_T,\beta,\sigma}$ has as index a *relation symbol interpretation* σ, mapping every relation symbol $r \in R$ with $\delta(r) = (s_1, s_2, \ldots, s_n)$ to a relation $\sigma(r) \subseteq \mathcal{I}(s_1) \times \mathcal{I}(s_2) \times \cdots \times \mathcal{I}(s_n)$. A model (or a solution) for a CHC C is an interpretation σ such that $\mathit{val}_{M_T,\beta,\sigma}(C) = \top$ for all $C \in \mathbb{C}$. For simplicity we only consider theories in which M_T is a singleton and omit it in the equations. We use the notation $\sigma \models_\beta \mathbb{C}$ if σ is a model of $\mathbb{C}$. A set of CHCs is *satisfiable* if it has a model, and *unsatisfiable* otherwise.

Computing the least model of a set of CHCs The least model σ_μ of a satisfiable set of CHCs $\mathbb{C}$ can be derived by computing a sequence of interpretations, until a fixed-point is reached. The sequence before reaching the fixed-point represents an under-approximation of the least model. In the context of program verification, the least model corresponds to the collecting semantics of a program.

The fixed-point computation starts with $\sigma^0 = \{r \mapsto \emptyset \mid r \in R\}$. Given some interpretation σ^n, the next interpretation $\sigma^{n+1} = T_{\mathbb{C}}(\sigma^n)$ is computed by applying the *immediate consequence operator* $T_{\mathbb{C}}$:

$$T_{\mathbb{C}}(\sigma)(r) = \left\{ \mathit{val}_{\beta,\sigma}(\bar{x}) \;\middle|\; \begin{array}{l} (H \leftarrow c \wedge c_h \wedge B_1 \wedge \ldots \wedge B_n) \in \mathbb{C}, \\ H = r(\bar{x}), \beta \text{ is a variable mapping s.t.} \\ \mathit{val}_{\beta,\sigma}(c \wedge c_h \wedge B_1 \wedge \ldots \wedge B_n) = \top \end{array} \right\} \tag{9}$$

The immediate consequence operator $T_{\mathbb{C}}$ is monotonic, i.e., if $\sigma_1 \subseteq \sigma_2$, then $T_{\mathbb{C}}(\sigma_1) \subseteq T_{\mathbb{C}}(\sigma_2)$, and the domain of $T_{\mathbb{C}}$ is a complete lattice (i.e., $R \times \mathcal{P}(D^*)$), which together imply that a least fixed-point exists by Tarski's fixed-point theorem [36]. This fixed-point computation does not necessarily reach a fixed-point after finitely many steps, as is the case in the example below. In general, the least model is $\bigcup_{n \in \mathbb{N}} T^n(\emptyset)$, the limit of the sequence $\sigma^0, \sigma^1, \ldots$.

Table 1: Theory of heaps operations and their interpretations as defined in [13]. The sorts are interpreted as $\mathcal{I}(\mathit{Heap}) = \mathcal{I}(\mathit{Object})^*$ and $\mathcal{I}(\mathit{Address}) = \mathbb{N}$.

Operation	**Signature**	**Interpretation**
nullAddress	$() \to \mathit{Address}$	0
emptyHeap	$() \to \mathit{Heap}$	ϵ
allocate	$\mathit{Heap} \times \mathit{Object} \to \mathit{Heap} \times \mathit{Address}$	$\langle h +\!\!+ [o], \lvert h\rvert + 1\rangle$
valid	$\mathit{Heap} \times \mathit{Address} \to \mathit{Bool}$	$0 < a \leq \lvert h\rvert$
read	$\mathit{Heap} \times \mathit{Address} \to \mathit{Object}$	$\begin{cases} h[a-1] & \text{if } 0 < a \leq \lvert h\rvert, \\ \mathit{defObj} & \text{otherwise.} \end{cases}$
write	$\mathit{Heap} \times \mathit{Address} \times \mathit{Object} \to \mathit{Heap}$	$\begin{cases} h[a-1 \mapsto o] & \text{if } 0 < a \leq \lvert h\rvert, \\ h & \text{otherwise.} \end{cases}$

Example. Consider the following set of CHCs $\mathbb{C}$:

$$\begin{aligned} r(1) &\leftarrow \top \\ r(x+1) &\leftarrow r(x) \wedge x \geq 0 \\ \bot &\leftarrow r(x) \wedge x < 0 \end{aligned}$$

To compute the least model for $\mathbb{C}$, we apply the immediate consequence operator $T_{\mathbb{C}}$ iteratively, starting from the empty interpretation $\sigma^0 = \{r \mapsto \emptyset\}$:

- $\sigma^1 = T_{\mathbb{C}}(\sigma^0) = \{r \mapsto \{1\}\}$
- $\sigma^2 = T_{\mathbb{C}}(\sigma^1) = \{r \mapsto \{1, 2\}\}$
- $\cdots$
- $\sigma^i = T_{\mathbb{C}}(\sigma^{i-1}) = \{r \mapsto \{1, \ldots, i\}\}$

This sequence of interpretations forms a chain, and its least fixed-point $\{r \mapsto \mathbb{N}^+\}$ represents the least model of $\mathbb{C}$. Any other model that satisfies $\mathbb{C}$ must include the least model. For instance $\sigma^{\mathbb{N}} = \{r \mapsto \mathbb{N}\}$ is a model of $\mathbb{C}$ too, but it is not the least one because $\mathbb{N}^+ \subset \mathbb{N}$. □

4.4 The theory of heaps

The operations and sorts of the theory of heaps, along with their interpretations, are provided in Table 1. The *Heap* sort is interpreted as a sequence of *Object*s, and the *Address* sort is interpreted as the set of natural numbers. The *Object* sort is for objects to be put on the heap, which often needs to be an ADT that refers to the *Address* sort (as in the example in Section 2), meaning it must be part of the heap theory declaration. The default object (*defObj*) returned on invalid reads is also define when declaring the theory, and is a term of sort *Object*. For a formal presentation of the theory, see [14], and for a decision procedure based on the semantics in Table 1, we refer to [13].

5 Heap Invariants for CHCs

5.1 Overview

We now formally introduce the transformation that augments a set of CHCs with heap invariants, based on the concept of space invariants [23]. We focus first on the simplest possible encoding without any of the refinements discussed in [23]. Due to the CHC setting (as opposed to Java), there are some key differences: (i) in our transformation, the original heap terms and operations are preserved, whereas the space invariant encoding eliminates the heap completely; and (ii) we use a single invariant symbol to describe the reachable states of all heap objects, whereas in space invariants a separate symbol is used per program type. Point (ii) stems from the fact that addresses in the theory of heaps are untyped, as the theory of heaps allows terms with a single user-defined *Object* sort to be placed on the heap. The type of an accessed object on the heap is therefore, in general, unknown at encoding time.

We define our transformation in rule-notation, with each of the heap operations being handled by one rule. Rules have the shape

$$\frac{\textit{CHC before translation} \tag{10}}{\textit{CHCs after transformation} \tag{11}}$$

and state that CHC (10) is replaced by the CHCs (11). The transformation runs through a set of CHCs only once.

5.2 Assumptions

Our transformation relies on three assumptions about the clauses to be rewritten. The assumptions can naturally be satisfied by CHCs that encode programs, and could be lifted in the transformation, at the cost of a more involved presentation.

Assumption 1: Isolated heap operations. We only consider CHCs that contain at most one heap operation from $F_h = \{\mathsf{allocate}, \mathsf{write}, \mathsf{read}, \mathsf{emptyHeap}\}$:

$$H(\bar{x}_0) \leftarrow B_1(\bar{x}_1) \wedge \cdots \wedge B_n(\bar{x}_n) \wedge c \wedge c_h \tag{12}$$

Here $\bar{x_0}, \ldots, \bar{x_n}$ denotes tuples of variables, c is a constraint not containing any of F_h, and c_h is either $\top$ or an equality of the form $\bar{x} = f(\bar{y})$, where $f \in F_h$ and $\bar{x}$, $\bar{y}$ are tuples of variables. Any set of CHCs can be normalised to the above form by splitting CHCs with multiple heap operations.

Assumption 2: Heap flowing from body. For a CHC as in (12), with c_h being of the form $\bar{x} = f(\bar{y})$ for some $f \in F_h$, we require that any heap variable h occurring in $\bar{y}$ also occurs in $\bar{x}_1, \ldots, \bar{x}_n$. This means that a heap to be updated has to come from the body of the CHC, while the updated heap may flow both to the clause body or to the clause head. We disallow CHCs such as $H(h) \leftarrow B(h', a, o), \mathsf{write}(h, a, o) = h'$ with flow from the head to the body.

Assumption 3: No heap out of thin air. For a CHC as in (12), we require that every heap variable h occurring in $\bar{x}_0$ also occurs in $\bar{x}_1, \ldots, \bar{x}_n$ or in the left-hand side of the equality c_h. This prevents uninitialized heap variables, which could have arbitrary content, from being passed to the head of a clause, and ensures that every object read from a heap has to be written to the heap in some other clause.

5.3 Transformation of heap updates

The theory of heaps has three update operations: emptyHeap, allocate and write. For the latter two operations, our transformation adds assertions that the heap invariant holds for the new object put on the heap. No transformation is necessary for emptyHeap, as the empty heap does not contain any objects.

emptyHeap The transformation rule for emptyHeap is given in (13) – (14). This rule preserves the original clause and does not add any new clauses.

$$\frac{head \leftarrow body \wedge h = \mathsf{emptyHeap}() \tag{13}}{head \leftarrow body \wedge h = \mathsf{emptyHeap}() \tag{14}}$$

allocate The transformation rule for allocate is given in (15) – (17). This rule preserves the original clause. Since an allocate operation always returns a valid heap–address pair, a valid predicate is not needed in (17) for memory safety.

$$\frac{head \leftarrow body \wedge (h_1, a) = \mathsf{allocate}(h_0, o) \tag{15}}{\begin{aligned} head &\leftarrow body \wedge (h_1, a) = \mathsf{allocate}(h_0, o) && (16)\\ I(a, o) &\leftarrow body \wedge (h_1, a) = \mathsf{allocate}(h_0, o) && (17)\end{aligned}}$$

write The transformation rule for write is given in (18) – (20). The rule is applied when $h_0 \in vars(body) \wedge h_1 \in vars(head)$. This rule also preserves the original clause, but unlike allocate, a write might be to an invalid address, so in ((20) the heap invariant is only asserted when the write is valid.

$$\frac{head \leftarrow body \wedge h_1 = \mathsf{write}(h_0, a, o) \tag{18}}{\begin{aligned} head &\leftarrow body \wedge h_1 = \mathsf{write}(h_0, a, o) && (19)\\ I(a, o) &\leftarrow body \wedge \mathsf{valid}(h_0, a) && (20)\end{aligned}}$$

5.4 Transformation of heap reads

The theory of heaps provides the operation read for reading. When the read address is valid,the heap invariant I constrains objects read from that address to only those that the invariant was *asserted* with (i.e., I appears at the head

of the CHC) in (17) and (20). When the read address is invalid, the read value defaults to *defObj* as shown in (23), according to the semantics of read.

$$\mathit{head} \leftarrow \mathit{body} \wedge o = \mathsf{read}(h, a) \tag{21}$$

$$\mathit{head} \leftarrow \mathit{body} \wedge o = \mathsf{read}(h, a) \wedge \mathsf{valid}(h, a) \wedge I(a, o) \tag{22}$$

$$\mathit{head} \leftarrow \mathit{body} \wedge o = \mathsf{read}(h, a) \wedge o = \mathit{defObj} \wedge \neg\mathsf{valid}(h, a) \tag{23}$$

6 Correctness of the Heap Invariants Transformation $\mathfrak{T}$

Let $\mathbb{C}_1$ be a set of CHCs, $\mathfrak{T}$ a transformation function over a set of CHCs and $\mathbb{C}_2 = \mathfrak{T}(\mathbb{C})$. We say that $\mathbb{C}_1$ and $\mathbb{C}_2$ are *equisatisfiable* if $\mathbb{C}_1$ is satisfiable if and only if $\mathbb{C}_2$ is satisfiable; in this case, $\mathfrak{T}$ is *correct*.

$$\text{There is } \sigma_1 \text{ s.t. } \sigma_1 \models_\beta \mathbb{C}_1 \implies \text{there is } \sigma_2 \text{ s.t. } \sigma_2 \models_\beta \mathbb{C}_2. \tag{24}$$

$$\text{There is } \sigma_2 \text{ s.t. } \sigma_2 \models_\beta \mathbb{C}_2 \implies \text{there is } \sigma_1 \text{ s.t. } \sigma_1 \models_\beta \mathbb{C}_1. \tag{25}$$

Equation (25) corresponds to the *soundness* of the transformation, and Equation (24) corresponds to its *completeness*. We will first show completeness in Theorem 1, and then soundness in Theorem 2.

6.1 Completeness of $\mathfrak{T}$

Theorem 1. *If $\mathbb{C}_1$ is satisfiable, then $\mathbb{C}_2$ is satisfiable.*

Proof. This direction of the proof is trivially established by interpreting I as $I = \{(a, o) \mid a \in \mathcal{I}(\mathit{Address}) \wedge o \in \mathcal{I}(\mathit{Object})\}$, i.e., the set of all (*Address*, *Object*) pairs. A way to see this is to go through the transformation rules and rewrite all atoms with the predicate I as $\top$, which leads to $\mathbb{C}_1 \equiv \mathbb{C}_2$.

6.2 Soundness of $\mathfrak{T}$

We show soundness through transfinite induction over the least-model computation of $\mathbb{C}_2$. First, we define an *induction formula* Φ that always holds during the least-model computation of $\mathbb{C}_2$ under a fixed structure M.

We collect all pairs (r, i) where r is a predicate symbol in $\mathbb{C}_2$ and i is the index of an argument of sort *Heap*:

$$HP = \{(r, i) \mid r \in \mathit{Rels}(\mathbb{C}_2) \wedge \delta(r)_i = \mathit{Heap}\}$$

Then the induction formula Φ is defined as:

$$\Phi = \bigwedge_{(r,i) \in HP} \forall \bar{x}.\ \big(r(\bar{x}) \rightarrow \forall a.\ (\mathsf{valid}(\bar{x}_i, a) \rightarrow I(a, \mathsf{read}(\bar{x}_i, a)))\big) \tag{26}$$

which asserts that the heap invariant I tracks all objects that can be read from all *valid* heap – address pairs.

Lemma 1. $\sigma^j = T^j_{\mathbb{C}_2}(\emptyset) \implies \sigma^j \subseteq T_{\mathbb{C}_2}(\sigma^j)$.

Lemma 1 asserts that the immediate consequence operator is increasing when the computation starts with the empty interpretation $\emptyset$ (i.e., the interpretation that maps all relation symbols to $\emptyset$).

Proof. σ^j denotes the result of applying $T_{\mathbb{C}_2}$ at step j (passing the result of the previous iteration as argument each time), starting with $\sigma^0 = \emptyset$. We have that $\emptyset \subseteq T_{\mathbb{C}_2}(\emptyset)$, and it follows from the monotonicity of $T_{\mathbb{C}_2}$ that its continued application will generate the ascending Kleene sequence $\emptyset \subseteq \sigma^1 \subseteq \ldots \subseteq \sigma^j$.

Lemma 2. *Φ holds under the least model σ_μ of $\mathbb{C}_2$ (i.e., $\sigma_\mu \models_\beta \Phi$).*

Proof. We do a transfinite induction over the fixed-point computation σ_μ of $\mathbb{C}_2$:

$$\sigma_\mu = \mu.T_{\mathbb{C}_2} \tag{27}$$

Base case: $\sigma^0 \models_\beta \Phi$. The base case vacuously holds, because σ^0 is the interpretation that maps all relation symbols to $\emptyset$, so $val_{\beta,\sigma}(r(\bar{x})) = \bot$ for all r and $\bar{x}$. This makes the left-hand side of the first implication in (26) false, which implies $\sigma^0 \models_\beta \Phi$.

Inductive step: $\sigma^j \models_\beta \Phi \implies \sigma^{j+1} \models_\beta \Phi$.

Assume the induction hypothesis $\sigma^j \models_\beta \Phi$. An application of the immediate consequence operator to σ^j yields the next interpretation, i.e., $\sigma^{j+1} = T_{\mathbb{C}_2}(\sigma^j)$. Recall the immediate consequence operator $T_{\mathbb{C}_2}$ (over $\mathbb{C}_2$):

$$T_{\mathbb{C}_2}(\sigma)(r) = \left\{ val_{\beta,\sigma}(\bar{x}) \;\middle|\; \begin{array}{l} (r(\bar{x}) \leftarrow c \wedge c_h \wedge B_1 \wedge \ldots \wedge B_n) \in \mathbb{C}_2, \\ \beta \text{ is a variable mapping s.t.} \\ val_{\beta,\sigma}(c \wedge c_h \wedge B_1 \wedge \ldots \wedge B_n) = \top \end{array} \right\} \tag{28}$$

The normal form of CHCs (which we assume for $\mathbb{C}_1$ and which is preserved by $\mathfrak{T}$) implies that there can be at most one heap term in a CHC that is produced as the result of a heap operation. By the induction hypothesis, this reduces the big conjunction in (26) to the case of checking (r, i) where r is a relation symbol with a heap argument, and i is the index of the heap term that was produced as a result of a heap operation. Furthermore, the only heap operations that produce a new heap term are $\mathsf{emptyHeap}$, $\mathsf{allocate}$ and write, so we only need to consider the rules over these operations and the unbound heap terms rule. Other (r, i) pairs satisfy Φ due to the induction hypothesis.

The proof is done in cases for rules involving aforementioned heap operations. In each case we show that $T_{\mathbb{C}_2}(\sigma^j) \models_\beta \Phi$.

Since $\mathbb{C}_2$ is the result of applying $\mathfrak{T}$ to a set of clauses $\mathbb{C}_1$ in normal form, a clause $r(\bar{x}) \leftarrow c \wedge c_h \wedge B_1 \wedge \ldots \wedge B_n$ must be in one of the following forms (where c_h is either $\top$ or an allowed heap formula in the normal form):

1. $c_h \triangleq h = \mathsf{emptyHeap}()$ due to (14).
   By the axioms of the theory of heaps, $h = \mathsf{emptyHeap}() \rightarrow \forall a.\ \neg\mathsf{valid}(h, a)$. Due to the contradiction in the left-hand side of the second implication $\mathsf{valid}(\bar{x}_i, a)$, this case shows $\sigma^{j+1} \models_\beta \Phi$.

2. $c_h \triangleq (h_1, a_1) = \mathsf{allocate}(h_0, o)$ and $r(\bar{x}) \not\equiv I(a, o)$ due to (16).
   By the induction hypotheses, we know that $\forall a.\ \mathsf{valid}(h_0, a) \rightarrow I(a, \mathsf{read}(h_0, a)$ holds. We also know, by the axioms of the theory of heaps, that after the allocation $\mathsf{valid}(h_1, a_1) \wedge \mathsf{read}(h_1, a_1) = o$ holds. Due to (17), $I(a_1, o) \leftarrow c \wedge c_h \wedge B_1 \wedge \ldots \wedge B_n$ is also in $\mathbb{C}_2$. Due to $T_{\mathbb{C}_2}$, this implies that $val_{\beta,\sigma}((a_1, o)) \in \sigma^{j+1}(I)$ with $o = \mathsf{read}(h_1, a_1)$. By Lemma 1, this shows $\sigma^{j+1} \models_\beta \Phi$.
3. $c_h \triangleq h_1 = \mathsf{write}(h_0, a, o)$ and $r(\bar{x}) \not\equiv I(a, o)$ due to (19).
   The formula $\forall a'.\ \mathsf{valid}(h_0, a') \rightarrow I(a, \mathsf{read}(h_0, a')$ holds by the induction hypotheses. We also know, by the axioms of the theory of heaps, that $(\mathsf{valid}(h_0, a) \rightarrow o = \mathsf{read}(h_1, a)) \wedge (\neg\mathsf{valid}(h_0, a) \rightarrow h_0 = h_1)$. We consider both sides of this conjunction in cases. (i) Assume that $\neg\mathsf{valid}(h_0, a)$ holds. Since $h_0 = h_1$, by the induction hypotheses $\sigma^{j+1} \models_\beta \Phi$ holds. (ii) Assume that $\mathsf{valid}(h_0, a)$ holds. We have $o = \mathsf{read}(h_1, a)$. Due to (20), $I(a, o) \leftarrow \mathsf{write}(h_0, a, o) \wedge \mathsf{valid}(h_0, a)$ is also in $\mathbb{C}_2$. Due to $T_{\mathbb{C}_2}$, this implies that $val_{\beta,\sigma}((a, o)) \in \sigma^{j+1}(I)$ with $o = \mathsf{read}(h_0, a)$. By Lemma 1, this shows $\sigma^{j+1} \models_\beta \Phi$.

CHCs in other forms do not generate or modify a heap term, and $\sigma^{j+1} \models_\beta \Phi$ directly holds by the induction hypothesis and Lemma 1.

Limit case: $\forall j < \lambda.\ \sigma^j \models_\beta \Phi \implies \sigma^\lambda \models_\beta \Phi$.

Assume the induction hypothesis holds for all $j < \lambda$.

By the definition of the limit ordinal and the fixed-point computation in (27), σ^λ is the limit of the sequence $(\sigma^j)_{j<\lambda}$, i.e., $\sigma^\lambda = \bigcup_{j<\lambda} \sigma^j$.

For each predicate $r \in R$, consider a tuple of arguments $\bar{x}$ that include some *Heap* argument h. By the definition of σ^λ, if $r(\bar{x}) \in \sigma^\lambda$, then there exists a $j < \lambda$ such that $r(\bar{x}) \in \sigma^j$. Since the induction hypothesis holds for all $j < \lambda$, we have $\sigma^j \models_\beta \Phi$. Therefore, for each address a, if $\mathsf{valid}(h, a)$ holds, then by (26), we have $I(a, \mathsf{read}(h, a)) \in \sigma^j$. Hence, $I(a, \mathsf{read}(h, a)) \in \sigma^\lambda$, as σ^λ is the union of all σ^j for $j < \lambda$.

This shows $\sigma^\lambda \models_\beta \Phi$ for the limit case, concluding the proof of Lemma 2.

Theorem 2. *If $\mathbb{C}_2$ is satisfiable, then $\mathbb{C}_1$ is satisfiable.*

Proof. Since all rules except the rule for heap reads preserve the original CHCs from $\mathbb{C}_1$, satisfiability of those clauses remains the same in their least models, σ_1 for $\mathbb{C}_1$ and σ_2 for $\mathbb{C}_2$. As a result, we only need to show that the theorem holds in CHCs with heap reads.

Consider the case where $\mathbb{C}_1$ has a clause where the heap read rule given in (21) – (23) is applicable. In the least models of $\mathbb{C}_1$ and $\mathbb{C}_2$, we consider the two cases where $\mathsf{valid}(h, a)$ and $\neg\mathsf{valid}(h, a)$ hold. (i) Assume $\neg\mathsf{valid}(h, a)$. The body of (22) has a contradiction, which makes it true. Since (21) $\equiv$ (23), this case is proved. (ii) Next, assume $\mathsf{valid}(h, a)$. The body of (23) has a contradiction, which makes it true. By Lemma 2, we have that $\sigma_2 \models_\beta \Phi$ (where Φ is the induction formula for $\mathbb{C}_2$). In Φ (26), we assume r is a relation symbol of one of the body literals of (22) that has h as an argument, and that $\bar{x}_i = h$. By Lemma 2, and since $o = \mathsf{read}(h, a)$, we have that $I(a, o)$ holds. As a result, we have (22) $\equiv$ (21).

Showing that the CHCs with heap reads in both $\mathbb{C}_1$ and $\mathbb{C}_2$ are equivalent in both cases concludes the proof of Theorem 2.

We have proved (24) and (25) by Theorems 1 and 2 respectively, which concludes that the transformation $\mathfrak{T}$ is correct (i.e., $\mathbb{C}$ and $\mathfrak{T}(\mathbb{C})$ are equisatisfiable).

7 Extensions

Flow sensitivity. The heap invariant introduced by the transformation we presented is flow insensitive: it cannot distinguish between different updates to the same address. The space invariants presented in [23] are also flow insensitive, but the authors describe how it can be made flow sensitive through an extension. The same approach can be adapted to our transformation at the level of CHCs to make the heap invariant flow sensitive.

We first assign a unique tag to every CHC that contains allocate or write. Then, we redefine the *Object* sort to a *tagged Object*, which is a pair containing the original object and its tag. This has the effect of storing the tag of every update site in the object, and consequently in the heap and the heap invariant. In every CHC that contains a read, we introduce a constraint ϕ, specific to that read, such that the tag of the read object can only come from certain updates. As in [23], this extension requires an external oracle to determine ϕ.

Adding more arguments to the heap invariant. In some cases the heap invariant is insufficient to express the relations between heap updates and reads. For example, in Figure 1 at line 9, assume that the written value is changed from "3" to "`x > 0 ? y : z`" where `x`, `y` and `z` are program variables. A useful I cannot be expressed without referring to these variables, but more arguments can always be introduced to I without affecting the correctness of the transformation.

8 Conclusion

We have developed a sound and complete transformation of CHCs that generalises the concept of space invariants to the CHC level using the theory of heaps. By lifting the transformation from Java programs to CHCs, our approach becomes language-independent and can be implemented directly within CHC solvers. We provided formal proofs of correctness for both soundness and completeness, without assuming the absence of uninitialised memory accesses.

The heap invariants transformation facilitates the inference of universally quantified heap invariants, and provides a formally verified foundation for future extensions, such as making heap invariants flow-sensitive. The original heap constraints within the CHCs are preserved, which allows combining the heap invariants transformation with other approaches such as shape analysis and tree automata to allow verification of more complex heap-manipulating programs.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Ji, R., Rezine, A.: Shape analysis via monotonic abstraction. In: Muscholl, A., Ramanujam, R., Rusinowitch, M., Schwentick, T., Vianu, V. (eds.) Beyond the Finite: New Challenges in Verification and Semistructured Data, 20.04. - 25.04.2008. Dagstuhl Seminar Proceedings, vol. 08171. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2008), http://drops.dagstuhl.de/opus/volltexte/2008/1559/
2. Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. Acta Informatica **53**(4), 357–385 (2016). https://doi.org/10.1007/S00236-015-0235-0
3. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: Solcmc: Solidity compiler's model checker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 325–338. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_16
4. Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpachecker 2.3 with strategy selection - (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14572, pp. 359–364. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
6. 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
7. Bjørner, N.S., 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
8. Blicha, M., Britikov, K., Sharygina, N.: The golem Horn solver. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 209–223. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_10
9. Böhme, S., Moskal, M.: Heaps and data structures: A challenge for automated provers. In: Bjørner, N.S., Sofronie-Stokkermans, V. (eds.) Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw,

Poland, July 31 - August 5, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6803, pp. 177–191. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_15
10. Bouajjani, A., Dragoi, C., Enea, C., Rezine, A., Sighireanu, M.: Invariant synthesis for programs manipulating lists with unbounded data. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 72–88. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_8
11. Britikov, K., Zlatkin, I., Fedyukovich, G., Alt, L., Sharygina, N.: Soltg: A chc-based solidity test case generator. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 466–479. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_23
12. Ernst, G.: Korn - software verification with Horn clauses (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 559–564. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_36
13. Esen, Z., Rümmer, P.: Reasoning in the theory of heap: Satisfiability and interpolation. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12561, pp. 173–191. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_9
14. Esen, Z., Rümmer, P.: An SMT-LIB theory of heaps. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022. CEUR Workshop Proceedings, vol. 3185, pp. 38–53. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3185/paper1180.pdf
15. Esen, Z., Rümmer, P.: Tricera: Verifying C programs using the theory of heaps. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 380–391. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_45
16. Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 572–585. ACM (2017). https://doi.org/10.1145/3062341.3062382
17. Giet, J., Ridoux, F., Rival, X.: A product of shape and sequence abstractions. In: Hermenegildo, M.V., Morales, J.F. (eds.) Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14284, pp. 310–342. Springer (2023). https://doi.org/10.1007/978-3-031-44245-2_15
18. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-

24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
19. 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
20. Holík, L., Kotoun, M., Peringer, P., Soková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Bloem, R., Arbel, E. (eds.) Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10028, pp. 202–209 (2016). https://doi.org/10.1007/978-3-319-49052-6_13
21. Holík, L., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Fully automated shape analysis based on forest automata. 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. 740–755. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_52
22. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. Formal Methods Syst. Des. **57**(3), 343–400 (2021). https://doi.org/10.1007/S10703-021-00366-4
23. 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://doi.org/10.29007/zrct
24. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9779, pp. 352–358. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_19
25. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 17–34. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_2
26. Kuru, I., Gordon, C.S.: Safe deferred memory reclamation with types. In: Caires, L. (ed.) Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11423, pp. 88–116. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_4
27. Li, J., Lattuada, A., Zhou, Y., Cameron, J., Howell, J., Parno, B., Hawblitzel, C.: Linear types for large-scale systems verification. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–28 (2022). https://doi.org/10.1145/3527313
28. Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: Chc-based verification for rust programs. In: Müller, P. (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. Lecture Notes in Computer Science, vol. 12075, pp. 484–514. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_18
29. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962. pp. 21–28. North-Holland (1962)
30. Nicole, O., Lemerre, M., Rival, X.: Lightweight shape analysis based on physical types. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 219–241. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_11
31. Peringer, P., Soková, V., Vojnar, T.: PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 408–412. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
32. Podelski, A., Wies, T.: Boolean heaps. In: Hankin, C., Siveroni, I. (eds.) Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3672, pp. 268–283. Springer (2005). https://doi.org/10.1007/11547662_19
33. 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
34. S, S.P., Fedyukovich, G., Madhukar, K., D'Souza, D.: Specification synthesis with constrained Horn clauses. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 1203–1217. ACM (2021). https://doi.org/10.1145/3453483.3454104
35. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002). https://doi.org/10.1145/514188.514190
36. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2), 285 – 309 (1955), `https://doi.org/10.2140/pjm.1955.5.285`
37. Toubhans, A., Chang, B.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7737, pp. 375–395. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_23
38. Wesley, S., Christakis, M., Navas, J.A., Trefler, R.J., Wüstholz, V., Gurfinkel, A.: Inductive predicate synthesis modulo programs. In: Aldrich, J., Salvaneschi, G. (eds.) 38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria. LIPIcs, vol. 313, pp. 43:1–43:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik

(2024). https://doi.org/10.4230/LIPICS.ECOOP.2024.43, https://doi.org/10.4230/LIPIcs.ECOOP.2024.43