Efficient Modular SMT-Based Model Checking of Pointer Programs*

Isabel Garcia-Contreras[1] [0000−0001−6098−3895],
Arie Gurfinkel[1] [0000−0002−5964−6792], and Jorge A. Navas[2] [0000−0002−0516−1167]

[1] University of Waterloo, Waterloo ON, Canada
{igarciac,agurfink}@uwaterloo.ca
[2] Certora, USA
jorge@certora.com

Abstract. Modularity is indispensable for scaling automatic verification to large programs. However, modularity also introduces challenges because it requires inferring and abstracting the behavior of functions as *summaries* – formulas that relate the function's inputs and outputs. For programs manipulating memory, summaries must include the function's *frame*, i.e., how the content memory is affected by the execution of the function. In SMT-based model-checking, memory is often modeled with (unbounded) logical arrays and expressing frames generally requires universally quantified formulas. Such formulas significantly complicate inference and subsequent reasoning and are thus to be avoided. In this paper, we present a technique to encode the memory that is bounded explicitly, eliminating the need for quantified summaries. We build on the insight that the size of frames can be statically known. This enables replacing unbounded arrays with *finite maps* – a finite collection of key-value pairs. Specifically, we develop a new static analysis to infer the finite parts of a function's frame. We then extend the theory of arrays to the theory of finite maps and show that satisfiability of Constrained Horn Clauses (CHCs) over finite maps is reducible to satisfiability of CHCs over the base theory. Finally, we propose a new encoding from imperative programs to CHCs that uses finite maps to model explicitly the finite memory passed in function calls. The result is a new verification strategy that preserves the advantages of modularity while reducing the need for quantified frames. We have implemented this approach in SeaHorn, a state-of-the-art CHC-based software model checker for LLVM. An evaluation on Linux Drivers from SV-COMP shows the effectiveness of our technique.

Keywords: Modular Verification · Software Model Checking · Constrained Horn Clauses · Pointer Analysis

* Part of this work was done when third author worked for SRI International and first author was visiting him. The work was partially funded by FPU grant 16/04811, MICINN project PID2019-108528RB-C21 *ProCode*, and the Comunidad de Madrid P2018/TCS-4339 *BLOQUES-CM* program.

1 Introduction

Modularity is indispensable for scaling automatic verification, such as software model checking. Reasoning modularly about a program involves abstracting the behavior of its functions in the form of a summary. For programs manipulating memory inferring summaries can be especially challenging. The reason is that summaries need to express the *frame* of the function, i.e., how the function modifies memory in any execution.

We focus on automated modular program verification using Constrained Horn Clauses (CHCs). In this setting, program verification is reduced to satisfiability of a set of logical rules (or clauses) [6], where unknown predicates represent summaries and inductive invariants. Satisfiability of CHCs is in general undecidable but, in practice, it is solved using so-called CHC solvers (e.g., HoIce [8], and Spacer [16]). CHC solvers automatically synthesize inductive invariants and, in the case of modular verification, function summaries.

In CHCs, memory side-effects are encoded by first *purifying* program statements to make such side-effects explicit, and then, encoding memory content by (unbounded) logical arrays. Each summary predicate relates arrays representing input and output memory contents. While this encoding is simple to implement, it is challenging to solve because it requires the CHC solver to discover function frames, that are typically expressed using quantified formulas. Although reasoning with quantified formulas is supported by some CHC solvers (e.g., [12]), it remains very challenging and is best to be avoided whenever possible.

Quantifiers are needed to restrict arrays at an unbounded number of indices. This is required to express how the execution of a function affects the state of memory. A key observation is that modeling the finite parts of a function's memory does not require the full power of arrays. The memory that is finitely accessed can be modeled using only scalar variables, avoiding the need for quantifiers. In this paper, we present a fully automatic CHC encoding of C programs that alleviates the problem of quantified frames based on this observation.

First, we introduce a new static analysis to compute: (a) which memory regions used in a function are accessed only finitely by it, (b) how many bytes are accessed per region, and (c) what are all the access paths for the finitely accessed memory. Our analysis is based on an existing alias analysis that ensures the soundness of our approach. Second, we model bounded memory, i.e., finite associative arrays, within SMT. For this, we propose a *new* SMT theory of *finite maps*. Finite maps modify the theory of arrays to account for a fixed number of key-value pairs. We show that the theory of finite maps is reducible to underlying SMT theories, and extend the reduction to CHCs (i.e., reduce CHCs with finite maps, to CHCs without). Finally, we extend the CHC encoding of SeaHorn to use finite maps for finite memory regions passed to functions. The key difficulty is in the handling of call sites since they must explicitly express the frame conditions.

We implemented our encoding using SeaHorn and evaluated it on Linux Drivers from SV-COMP. We show that the new encoding improves the original, array-based, one of SeaHorn. However, we also noticed that arrays sometimes provide a beneficial abstraction. Therefore, we relax our encoding to allow mixing arrays and finite maps for best performance.

2 Related work

The frame problem is a well-known problem in artificial intelligence [20] and program analysis. In this section, we discuss the related work in the areas of deductive verification and model checking.

Deductive verification. Including the footprint of a function in its specification to deal with the frame problem is a common solution in deductive verification. This is done explicitly or implicitly. An example of explicit footprints is dynamic frames [15] and the `reads` and `modifies` annotations in Dafny [19]. Examples of implicit footprints are implicit dynamic frames [27], permissions [22], and Separation Logic [26]. Explicit approaches describe the heap using additional assertions in the base logic, while implicit approaches embed heap information in the assertions by extending the logic. These have been proven difficult to integrate into SMT-based software model checkers, due to the difficulty of using SMT solvers to reason about both heap shape and content (Piskac et al. [23]). Our approach can be seen as computing the footprint explicitly but partitioning it into bounded and unbounded. The footprint is computed automatically, similar in spirit to how procedure specifications are inferred in tools such as Infer [7]. Most significantly, our approach is tightly integrated with automatic invariant inference over the content of the heap.

Inlining-based model checking. Tools based on bounded model checking (e.g., CBMC [9], LLBMC [21], and SMACK [25]) inline all procedures, which avoids the frame problem. Inlining is also implemented by unbounded tools such as UFO [2], SeaHorn [10], CPAChecker [5], and UAutomizer [13].

Summary-based model checking. Unbounded model checkers such as CPAChecker, Whale [1], and UAutomizer use inter-procedural model checking techniques to compute procedure summaries. The technique proposed by Beyer and Friedberger [3] lifts the idea of *Block-Abstraction Memoization (BAM)* from basic blocks to procedure boundaries. Procedures can be analyzed by using any of the intra-procedural model checking algorithms available in CPAChecker. The technique then generates summaries and stores them in a cache for future reuse. Whale computes summaries by exploiting sequence interpolants generated from underapproximations (i.e., finite traces) of functions. Finally, UAutomizer relies on *Nested Interpolants* [14] to produce summaries but they depend on the calling context so they might be harder to reuse. Most importantly, none of these techniques tackle the problem of frame inference. Note also this paper does not propose a new inter-procedural model-checking algorithm. Instead, our goal is to improve the encoding of verification conditions to reduce the need for quantifiers in CHC solvers.

Modeling memory in SMT-based model checking. Most existing software model checkers use some form of purification. In all cases, memory is modeled as either arrays [10] or lambdas [21]. Sometimes a finite abstraction of memory is used (see e.g., Blast [4]) modeling precisely only a few levels of pointer deference (e.g., `*p` and `**p`). In contrast, our modeling is precise – we use finite footprint wherever possible and arrays only if necessary. While the need for a finite map theory for program reasoning has been identified before [17], we propose a theory of finite maps that is more suitable for encoding finite memory in CHCs.

3 Motivating Example

We illustrate our approach with an example. We begin with *purification*. Fig. 1a shows the definition of a data structure `S` with a field `x` in a C-like language and two functions over `S`: `init_x`, which stores the value `0` in the field `x`, and `read_x`, which returns the value of the field. In its purified version (Fig. 1b), memory operations are made explicit with a structure of type `Memory` (a special array) that represents an unbounded sequence of bytes. The signature of every function is extended to include a `Memory` parameter, and memory reads and writes are operations over it. Given a variable `MEM` of type `Memory`, and assuming that field `x` is at offset `0`, `s->x = v` is encoded as `MEM[s] = v`, and `s->x` as `MEM[s]`.

Consider the program defined by Figs. 1a and 1c. In Fig. 1c, the `main` procedure allocates two structures p and q of type `S` on lines 3 and 4. Line 6 models that the pointers p and q must be disjoint. Let us assume that after the execution of some arbitrary code the pointer analysis infers that p and q might alias (line 7). On line 10, some values are stored at `p->x` and `q->x`. Fig. 1d shows the purified version of Fig. 1c. Note that memory allocations do not change the state of `MEM`. In this example, the property to be verified is `assert(read_x(q, &MEM) == 20)` (line 13). The semantics of the program together with this property is encoded by the following CHCs[3]:

$$r = m[s] \rightarrow \mathit{read\_x}(s, r, m) \qquad \text{(CHC 1)}$$

$$m_2 = m_1[s \leftarrow 0] \rightarrow \mathit{init\_x}(s, m_1, m_2) \qquad \text{(CHC 2)}$$

$$p + 4 < q \ \wedge m_1 = m[p \leftarrow 10] \wedge m_2 = m_1[q \leftarrow 20] \wedge \qquad \text{(B3a)}$$

$$\mathit{init\_x}(p, m_2, m_3) \wedge \mathit{read\_x}(q, r, m_3) \rightarrow r = 20 \qquad \text{(CHC 3)}$$

The summaries computed for `read_x` and `init_x` need to be precise enough to prove the satisfiability of CHC 3. For `read_x`, referring to the content of *one* memory location is enough: $\lambda s, r, m.\ r = m[s]$. Since m_3 is an argument of `init_x`, its summary needs to express how m_3 is related to m_2, i.e., how memory is updated:

$$\lambda p, m_2, m_3.\ m_3[p] = 0 \wedge \forall i \neq p.\ m_3[i] = m_2[i]$$

The first conjunct expresses the memory location that is modified by `init_x`, and the second expresses the frame, using a quantified formula.

We now show how a manual transformation in the C program eases the verification task by eliminating the need of inferring quantified summaries. Consider

[3] We use the syntax $a[i]$ and $a' = a[i \leftarrow v]$ to denote, respectively, an array select at index i and an array store at index i with value v.

```c
typedef struct S { int x; } S;
void init_x(S *s) {
  s->x = 0;
}
int read_x(S *s) {
  return s->x;
}
```

(a)

```c
typedef struct S { int x; } S;
void init_x(S *s, Memory *MEM) {
   (*MEM)[s] = 0;
}
int read_x(S *s, Memory *MEM) {
  return (*MEM)[s];
}
```

(b) Purified functions from Fig. 1a

```c
void main() {

  S* p = malloc(sizeof(S));
  S* q = malloc(sizeof(S));
  // Model part of malloc semantics
  assume(p + sizeof(S) < q);
  // Code makes the analyzer think
  // that p and q alias

  p->x = 10;   q->x = 20;

  init_x(p);
  assert(read_x(q) == 20);
}
```

(c)

```c
void main() {
  Memory MEM;
  S* p = malloc(sizeof(S));
  S* q = malloc(sizeof(S));
  // Model part of malloc semantics
  assume(p + sizeof(S) < q);
  // Code makes the analyzer think
  // that p and q alias

  MEM[p] = 10;    MEM[q] = 20;

  init_x(p, &MEM);
  assert(read_x(q, &MEM) == 20);
}
```

(d) Purified program from Fig. 1c

```c
void main() {

  S* p = malloc(sizeof(S));
  S* q = malloc(sizeof(S));
  // Model part of malloc semantics
  assume(p + sizeof(S) < q);
  // Code makes the analyzer think
  // that p and q alias

  p->x = 10;    q->x = 20;

  S tmp;
  tmp.x = p->x;
  init_x(&tmp);
  p->x = tmp.x;
  assert(read_x(q) == 20);
}
```

(e)

```c
void main() {
  Memory MEM;
  S* p = malloc(sizeof(S));
  S* q = malloc(sizeof(S));
  // Model part of malloc semantics
  assume(p + sizeof(S) < q);
  // Code makes the analyzer think
  // that p and q alias

  MEM[p] = 10;    MEM[q] = 20;

  S tmp;   Memory AUX;
  AUX[&tmp] = MEM[p];
  init_x(&tmp, &AUX);
  MEM[p] = AUX[&tmp];
  assert(read_x(q, &MEM) == 20);
}
```

(f) Purified program from Fig. 1e

Fig. 1: Some functions (left) and their purified versions (right)

the program defined by Figs. 1a and 1e. The `main` function differs from Fig. 1c in that a new structure `tmp` is passed to `init_x`. The content of `p->x` is stored in `tmp.x` before calling `init_x`, and `tmp.x` is copied back to `p->x` right after the call returns. After purification (Fig. 1f), before the call, the memory contents accessed by the callee (`MEM[p]`) are copied into a new memory `AUX`, because the content `tmp` and `p` is known to be stored in different memory regions. After the call, the contents are copied back from `AUX` into `MEM`. It is not hard to see that the programs in Figs. 1d and 1f are equivalent. However, the latter is much easier to verify. The program in Figs. 1b and 1f is encoded by CHCs $\{1, 2\}$ and:

$$B3a \wedge aux_1 = aux[tmp \leftarrow m_2[p]] \wedge \tag{L4a}$$

$$\mathit{init\_x}(tmp, aux_1, aux_2) \ \wedge m_3 = m_2[p \leftarrow aux_2[tmp]] \wedge \tag{L4b}$$

$$\mathit{read\_x}(q, r, m_3) \rightarrow r = 20 \tag{CHC 4}$$

The difference between CHC 3 and CHC 4 is in the literals before and after the predicate call to `init_x`. In CHC 4, the array contents accessed by *init_x* are copied to a different array *aux* in L4a. The predicate *init_x* takes *aux* and *tmp* arguments instead of m_i, and finally, the values of *aux* are copied back to m. Note that CHCs $\{1, 2, 3\}$ and $\{1, 2, 4\}$ are *equisatisfiable*. However, the key advantage of CHCs $\{1, 2, 4\}$ is that the relation between m_2 and m_3 is explicit in CHC 4. Since *aux* arrays are not relevant to the property, the behavior of *init_x* can be abstracted with the trivial summary "true", which is not quantified.

This example showed how a manual transformation in the C program eases the verification task by eliminating the need of inferring quantified summaries. In the rest of the paper, we show how to encode automatically in the CHCs the idea behind this example, without any user intervention. This requires: (1) finding the finite memory footprint of a function (i.e., the candidates to be copied to an auxiliary variable) and (2) identifying the memory locations that are accessed, to copy their content to/from auxiliary memory objects.

Remarks. In this example, using auxiliary arrays to represent the finite memory accessed in `init_x` was enough to avoid a quantified summary. An alternative approach is to use partial array equalities from the *extensional* theory of arrays [29]. This, however, still uses arrays, and, therefore, does not eliminate the need for quantifiers. A more concise logic to represent finite memory is the theory of *finite maps*. In Sec. 5, we describe the theory of finite maps and how to extend CHCs with finite maps.

4 Static Analysis of Memory Footprints

The C memory model interprets a pointer as a pair (id, o) where id is an identifier that uniquely defines a memory object and o defines the byte in the object being pointed to. The number of objects is unbounded. Points-to analysis typically abstracts the unbounded set of concrete memory objects as a finite set of abstract objects (also called memory regions). A points-to analysis is sound if whenever a pointer p does not point to an abstract object, then there is no actual execution in which p points to any concrete object represented by the abstract object.

We rely on the Data Structure Analysis (DSA) of [11, 18] which provides a unification-based, context- and field-sensitive points-to analysis, that supports pointer arithmetic. In DSA, a pointer can only point to one abstract object due to its unification-based nature [28]. The analysis results are presented in the form of DSA graphs. A *DSA graph* is a triple (C, E, σ), where C is a finite set of abstract cells. Each *cell* is a pair of a memory region identifier and a byte

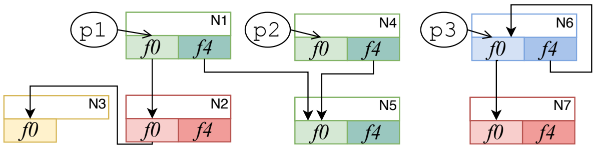

Fig. 2: Points-to graph of a function `foo(S *p1, S *p2, S *p3)`.

offset; $E \subseteq C \times C$ is a set of edges between cells, denoting points-to relations; and environment $\sigma : \mathit{Var} \mapsto C$ maps pointer variables to cells.

As part of the DSA analysis, a summary graph is built for each function. A *summary* graph contains all the memory objects accessed by the function and its callees, and their points-to relationships, i.e., its *memory footprint.* These graphs, called henceforth DSA graphs, are computed ignoring how and where the function is called, assuming that there is no aliasing between input parameters.

Example 1 (DSA graph). Fig. 2 shows a DSA graph generated from a function `foo` with parameters `p1`, `p2`, and `p3`. Each of the cells encodes an offset in the memory region that may be accessed during a concrete execution. For example, the memory object **N1** has 2 cells $f0$ and $f4$ (naming the offsets). This means that at some point of the execution of `foo` (or its callees), `p1->f0` and `p1->f4` may be accessed (read or written). The cells of **N2** are representing that `p1->f0->f0` and `p1->f0->f4` may be accessed. Since DSA graphs are over-approximations of the concrete memory used during any execution, the absence of a cell in the graph implies that a memory location is never accessed. For example, `p1->f0->f0->f4` is never accessed because there is not a field $f4$ in **N3**.

The goal is to determine which memory objects are bounded to make them explicit in the encoding. First, we define the paths in a DSA graph.

Definition 1 (A path in the DSA graph). *Let $g = (C, E, \sigma)$ be a graph.* A sequence of cells $[c_1, c_2, \ldots, c_k]$ is a path in g *iff for every c_i, c_{i+1}, $1 \leq i < k$:*

$$\exists x, y, n \cdot (c_i, (n, x)) \in E \land (c_{i+1} = (n, y)) \in C.$$

This is the standard definition of a path in a graph, modified to capture that when a memory object with id n is reachable by some path, all its fields (i.e., all the cells that have the same id n) are reachable as well. In Fig. 2, `p1` points to cell $(\mathbf{N1}, f0)$ but both fields, $f0$ and $f4$, are reachable. An *access path* is a base variable followed by a finite sequence of field accesses. That is, an access path is a pair $(\mathit{var}, \mathit{acc})$, where var is a variable of the function, and acc is either a sequence with a single cell or a path between two cells. E.g., the access path of the expression `p1->f0->f0` is $(p1, [(N2, f0), (N1, f0)])$[4].

One way to determine if a cell represents a finite number of concrete memory locations is by computing the set of paths in the DSA graph. Intuitively, a cell

[4] For conciseness and presentation purposes, we use $[x_1, \ldots, x_n]$ to refer to $\texttt{cons}(x_1, (\ldots \texttt{cons}(x_n, \texttt{nil})))$ and write the paths reversed.

that is reachable by n paths represents at most n distinct memory locations. The following definition describes a cell *being finitely accessed* in terms of its paths.

Definition 2 (Finitely accessed cells). *Let g be a graph with cells C and $C' \subseteq C$. A cell $c \in C$ is* finitely accessed from C' *if the number of paths from $c' \in C'$ to c is finite.*

This definition is based on paths starting from arbitrary cells in the graph. However, in practice, we are interested only in the cells pointed by the parameters of the function, because only those are reachable by callers. Finding the cells in a summary graph that meet the property of Def. 2 allows identifying the finitely accessed memory regions of a function assuming no aliasing relationships before the function call. However, cells that are distinct in the callee maybe the same in the caller. For example, in a call of the form `foo(s1,s1,s3)`, objects **N1** and **N4**, shown to be distinct in the summary graph of Fig. 2, are actually the same since the same pointer `s1` is passed as the first and second parameter of `foo`. Therefore, these aliasing relationships must be considered to produce a sound encoding. In the following, given the graph g of a function f the predicate $\texttt{alias}_{call}(c, c')$ is true if c and c' (cells in g), may be the same in a *call* to f. The predicate $\texttt{alias}_{call}$ induces an equivalence relation over the cells in the graph, where two cells are related if they are the same at the function call.

Definition 3 (Finitely accessed equivalence class). *Let g be a graph with cells C, $C' \subseteq C$, and $\texttt{alias}_{call}$ the aliasing relation of the cells. The* equivalence class of $c \in C$ is finitely accessed from C' *iff all the elements in the equivalence class are finitely accessed. That is, $\forall d \in C$ such that $\texttt{alias}_{call}(c, d)$, d is finitely accessed from C'.*

Def. 3 lifts Def. 2 to the equivalence classes defined by $\texttt{alias}_{call}$. The following example illustrates the concepts in Defs. 1, 2, and 3.

Example 2 (Bounded memory objects). Consider again the graph in Fig. 2 and a predicate $\texttt{alias}_{call}$ that is true iff the cells in the graph have the same color. For example, $\texttt{alias}_{call}((N1, f0), (N4, f0))$ and $\texttt{alias}_{call}((N1, f4), (N4, f4))$ are facts. First, we determine which cells are finitely accessed (Def. 2). Memory objects that have self-loops are not finitely accessed, as they have an unbounded number of access paths. For example, the cell (**N6**, $f0$) has access paths `p3->f0`, `p3->f4->f0`, `p3->f4->f4->f0`,. . . For the same reason, cells that are reachable from memory objects with self-loops are also unbounded. For example, cell (**N7**, $f0$) can be accessed by `p3->f0->f0`, `p3->f4->f0->f0`, `p3->f4->f4->f0->f0`,. . . Thus, **N6** and **N7** encode unbounded memory accesses. For the remaining objects, **N1** to **N5**, all the cells in the same equivalence class need to be finitely accessed. Consider all the cells in green: **N1**, **N4**, and **N5**. All three objects are finitely accessed, so the equivalence classes of (**N1**, $f0$) and (**N1**, $f4$) are finitely accessed. **N2** is finitely accessed but its cells are in the same equivalence class as the cells in **N7** (red). Since **N7** is not finitely accessed, the classes of the cells of **N2** are not finitely accessed. Last, even if the classes of the cells of **N2** are not finitely accessed, **N3** is finitely accessed because its parents are finitely accessed.

```
compFiniteAPs($g = (C, E, \sigma), \texttt{alias}, \mathit{fparams}$)
  $C' := \{\sigma(p) \mid p \in \mathit{fparams})$
  $U :=$ exploreGraph($g, C'$)
  $APs := \emptyset$
  for all $p \in \mathit{fparams}$ do
      $ap_0 = \texttt{nil}$
      recCompAPs($\sigma(p), g, U, \texttt{alias}, p, ap_0, APs$)
  return $APs$

recCompAPs($c, g, U, \texttt{alias}, p, ap, APs$)
  if $c \in U$ then return
  if not aliasesUbnd($c, U, \texttt{alias}$) then
      for all $fc \in \texttt{Fields}(c, g)$ do
          $APs := APs \cup \{(p, \texttt{cons}(fc, ap))\}$
  for all $lc \in \texttt{Links}(c, g)$ do
      $ap' = \texttt{cons}(c, ap)$
      recCompAPs($lc, g, U, \texttt{alias}, p, ap', APs$)

aliasesUbnd($c, U, \texttt{alias}$)
  return $\exists c' \in U.\texttt{alias}(c, c')$

exploreGraph($g = (C, E, \sigma), C'$)
  for all $c \in C$ do
      $color[c] := \texttt{white}$
  $U := \emptyset$
  for all $c \in C'$ do
      exploreCell($c, g, color, U$)
  return $U$;

exploreCell($c, g, color, U$)
  $color[c] := \texttt{grey}$;
  for all $d \in \texttt{Links}(c, g)$ do
      if $color[d] = \texttt{grey}$ then
          propagateUbnd($d, g, color, U$)
      else if $color[d] = \texttt{white}$ then
          exploreCell($d, g, color, U$)
  $color[c] := \texttt{black}$;

propagateUbnd($c, g, color, U$)
  $U := U \cup \{c\}$
  $color[c] := \texttt{black}$
  for all $d \in \texttt{Links}(c, g)$ do
      if not ($color[d] = \texttt{black}$ and $d \in U$) then
          propagateUbnd($d, g, color, U$)
```

Fig. 3: Algorithm to find finite memory objects and all their access paths.

We have shown intuitively how to determine if cells are reachable only by a *finite number of paths*. Fig. 3 shows the proposed algorithm to find the finite memory objects used by a function and their access paths. Access paths are used later to encode the memory passed to a function at a call. The entry point is compFiniteAPs(g, alias, *fparams*) which takes a DSA graph g, a relation of its cells alias, and the function parameters *fparams*. First, the set of cells, C', pointed by *fparams* is computed, which is the starting point for traversing the graph. The algorithm is split into two steps. The function exploreGraph computes the set of cells that have an unbounded number of paths in g. Second, recCompAPs computes all access paths to cells that belong to equivalence classes that are finitely accessed through paths starting from C'.

Function exploreGraph(g, C') is similar to standard cycle-detection algorithms. However, when a cycle is detected in a memory object, all the cells that are reached from that object are also stored as unbounded. In this function, *color* is a map from cells to exploration status, denoted with a color: white, grey, or black, respectively, not explored, exploring, and explored. U is the set of cells with an unbounded number of paths. Given a cell $c = (n, o)$ and a graph g, Links(c, g) denotes the set of cells that are reachable from any cell in the same region n, i.e., all the c_i such that there is an edge of the form $(n, \_) \to c_i$ in g. First, all the cells in g are marked as unexplored. Then, starting from every cell in C', the cell is marked as grey (*exploring*), and all the reachable cells in one step (given by Links) are explored. If the cell is currently being explored (grey), a cycle has been encountered and propagateUbnd is used to mark them. If the cell has not been explored yet, then it is explored. Once all the links of the cell have been explored, the cell is marked as explored (`black`). The function propagateUbnd marks as explored and stores in U the cell c and all cells reachable from c.

After exploration, recCompAPs$(c, g, U, \texttt{alias}, p, ap, APs)$ computes the set of access paths to cells in equivalence classes that represent bounded memory. The argument c is the cell to be processed, g is the graph, U is the set of cells in g that represent unbounded memory, `alias` determines the equivalence classes, i.e., which cells need to be considered together, p is the base variable of the access path, and ap is the path followed in the graph to access c. In the recursion, loops in the graph are avoided by checking U before exploring a cell. Equivalence classes are considered in aliasesUbnd, which determines if a cell belongs to the same class as an unbounded cell. $\texttt{Fields}(c, g)$ denotes the set of cells in the same region as c. That is, $\texttt{Fields}((n, o), g) = \{c' \mid c' = (n, \_) \text{ in the cells of } g\}$. If c does not alias with unbounded cells, all the fields are stored in APs, together with how they were reached in ap (line 11 in Fig. 3). Last, the `Links` of the cell are explored, adding c to the path in the recursive call (line 14).

Example 3 (Access paths to cells encoding finite memory). Given the graph of Fig. 2 the following access paths to cells with finite access paths are found:

Class of $(\mathbf{N1}, f0)$: $\{(p1, [(N1, f0)]), (p2, [(N4, f0)]), (p2, [(N5, f0), (N4, f4)])\}$
Class of $(\mathbf{N1}, f4)$: $\{(p1, [(N1, f4)]), (p2, [(N4, f4)]), (p2, [(N5, f4), (N4, f4)])\}$
Class of $(\mathbf{N3}, f0)$: $\{(p1, [(N3, f0), (N2, f0), (N1, f0)]\}$

Remark. The correctness of our approach relies on the fact that DSA graphs over-approximate both the length and the number of access paths in the concrete memory graph. This follows from the fact that DSA graphs simulate all possible concrete memory graphs [11].

5 Theory of Finite Maps

We model the contents of finitely accessed memory through finite maps. This resembles an SMT-LIB unbounded array in that the map can have arbitrary keys, and a finite sequence, in that the number of entries is fixed. While the need for such a structure for program reasoning has been identified before [17], no theory is provided in the SMT-LIB standard. In this section, we propose a theory of finite maps that is suitable for encoding finite memory footprints. Our key contribution is a reduction procedure from CHCs defined over finite maps and integers to CHCs only over integers.

A *finite map* is composed of a set of key-value pairs. Its *sort* is defined by the sort of the keys, the sort of the values, and the *size* of the finite map, i.e., the maximum number of key-value pairs that it can store. For simplicity of presentation, we restrict ourselves to a finite map of size 2 but our implementation supports finite maps of arbitrary size. Similarly, we assume that finite maps are

$$\overset{\textsc{get-over-fm}}{\frac{\mathtt{get}(\mathit{fm},k)}{\mathtt{toLmd}(\mathit{fm})(k)}}\quad \overset{\textsc{get-over-set-1}}{\frac{\mathtt{get}(\mathtt{set}(\mathit{fm},k,v),l)}{v}}\ (k=l)\quad \overset{\textsc{get-over-set-2}}{\frac{\mathtt{get}(\mathtt{set}(\mathit{fm},k,v),l)}{\mathtt{get}(\mathit{fm},l)}}\ (k\neq l)\quad \overset{\textsc{get-over-ite}}{\frac{\mathtt{get}(\mathtt{ite}(c,\mathit{fm},\mathit{fe}),k)}{\mathtt{ite}(c,\mathtt{get}(\mathit{fm},k),\mathtt{get}(\mathit{fe},k))}}$$

Fig. 4: Reduction rules.

$$\text{If } k_0 \neq k_1 \wedge l = k_0\text{:}$$

$$\boxed{\overset{\textsc{get-over-fm-1}}{\frac{\mathtt{get}([k_0 \mapsto v_0|k_1 \mapsto v_1],l)}{v_0}}\quad \overset{\textsc{set-over-fm-1}}{\frac{\mathtt{set}([k_0 \mapsto v_0|k_1 \mapsto v_1],l,w)}{[k_0 \mapsto w|k_1 \mapsto v_1]}}}$$

$$\overset{\textsc{set-over-fm-2}}{\frac{\mathtt{set}([k_0 \mapsto v_0|k_1 \mapsto v_1],l,w)}{[k_0 \mapsto \mathtt{ite}(l=k_0,w,v_0)|k_1 \mapsto \mathtt{ite}(l=k_1,w,v_1)]}}\quad \overset{\textsc{ite-over-fm}}{\frac{\mathtt{ite}(c,[k_0 \mapsto v_0|k_1 \mapsto v_1],f)}{[k_0 \mapsto \mathtt{ite}(c,v_0,\mathtt{get}(f,k_0))|k_1 \mapsto \mathtt{ite}(c,v_1,\mathtt{get}(f,k_1))]}}$$

Fig. 5: Additional rules for optimization.

of the form $[k_0 \mapsto v_0|k_1 \mapsto v_1]$[5]. We define two operations over finite maps: *get*, denoted by $\mathtt{get}(\mathit{fm},k)$, which stands for the value of map *fm* for key k, and *set*, denoted by $\mathtt{set}(\mathit{fm},k,v)$, which stands for the map obtained after writing v at key k in *fm*. These operations are well-formed whenever the key used in the operation is in the range of the map. That is, it matches a key of an already stored key-value pair or the map contains a key-value pair that has not been initialized yet, and thus, has no key assigned. We always ensure that expressions are well-formed by construction, thus, we do not provide a well-formedness check. For well-formed formulas, these operations satisfy the usual array axioms:

- congruence: $k = l \implies \mathtt{get}(\mathit{fm},k) = \mathtt{get}(\mathit{fm},l)$
- get-over-set (1): $k = l \implies \mathtt{get}(\mathtt{set}(\mathit{fm},k,v),l) = v$
- get-over-set (2): $k \neq l \implies \mathtt{get}(\mathtt{set}(\mathit{fm},k,v),l) = \mathtt{get}(\mathit{fm},l)$

Reduction procedure. Applying the rules in Fig. 4 exhaustively to a formula with finite maps results in an equisatisfiable formula without finite maps. No assumptions are made about how the keys within the map are related. The function `toLmd` transforms a finite map into a lambda term: $\mathtt{toLmd}([k_0 \mapsto v_0|k_1 \mapsto v_1]) = \lambda x.(\mathtt{ite}(x = k_0, v_0, v_1))$. We do not support extensionality because it is not needed in our encoding.

Optimizations. Fig. 5 defines rules for optimization for the cases in which information about the keys is available. The application of these rules can be used to update finite maps "in-place" during a sequence of *set* operations, which can avoid an exponential blow-up caused by introducing *ite* terms.

CHCs over finite maps. In general, a Constrained Horn Clause (CHC) is a first-order formula of the form $\forall V \cdot (\phi \wedge \bigwedge p_i(X^i_1,\ldots,X^i_{n_i}) \implies h(X^h_1,\ldots,X^h_n))$, where V are all the free variables, ϕ is a constraint in some background theory, p_i are n_i-ary predicates, and $p_i(X^i_1,\ldots,X^i_{n_i})$ applications of predicates to first-order terms. The antecedent of the implication is called the *body* and the consequent is called the *head*.

[5] A finite map variable can always be expressed in this form using the size in its sort.

CHCs over finite maps extend general CHCs by allowing finite maps to appear in both the constraint ϕ and in arguments to the predicates, and extending the background theory with finite maps. To reduce CHCs with finite maps to CHCs without them, we apply the rules from Fig. 4 and Fig. 5 exhaustively to remove fnite maps from ϕ. To eliminate finite maps from arguments, we expand each finite map argument to the scalars defining its keys and values. For example, if $F = [k_0 \mapsto E_0 | k_1 \mapsto E_1]$ with two key-value pairs, then all predicate applications $p(\ldots, F, \ldots)$, in bodies and heads, are expanded into $p(\ldots, k_0, E_0, k_1, E_1, \ldots)$.

6 A CHC encoding with Finite Maps

In this section, we show how to extend the CHC encoding within SeaHorn to model memory using finite maps. Roughly, SeaHorn takes as input a C program with assertions (expressing the properties of interest) and produces a set of CHCs. Each CHC captures the semantics of one or multiple basic blocks (sequence of instructions) [6]. Loops are modeled by recursive CHCs and function calls are encoded as predicate calls in the body of a CHC, representing the effects of the call. In general, a CHC is of the form:

$$\mathit{loc}_n(s, a_0) \land \mathit{fun}(s, a_0, a_1) \land \phi(s, a_1, a_2) \implies \mathit{loc}_m(s, a_2) \qquad (\mathrm{CHC}_A)$$

where every variable represents a vector of variables and is implicitly universally quantified. The symbols loc_n, loc_m, and fun are predicate names. This clause models how location loc_m in the program may be reached from location loc_n. The literal $\mathit{fun}(s, a_0, a_1)$ captures that there is a function call between the two locations, and ϕ encodes the semantics of all program statements other than function calls. s is a vector of scalar variables, and each a_i are array vectors that model the state of memory at the different locations. a_0 models the state at location n. It is passed to predicate fun, since it may modify memory, producing the next state a_1. The semantics of the remaining statements of the program, from loc_n to loc_m, is modeled by the constraint $\phi(s, a_1, a_2)$, with a_2 the state of the memory at loc_m, in the consequent of the clause. The number of variables in a_i is the number of disjoint memory regions discovered by the pointer analysis.

When encoding bounded memory regions as finite maps the cells that were identified to be bounded are represented using finite map terms, instead of arrays. In general, a clause with finite maps in our proposed encoding is of the form:

$$\begin{aligned} &\mathit{loc}_n(s, b_0, f_0) \land \mathit{fun}(s, b_0', f_{in}, b_1', f_{out}) \ \land \phi_A(b_0, f_{out}, b_1) \land \\ &\qquad \land \phi_{FM}(f_0, f_{out}, f_1) \ \land \phi(s, b_1, b_2, f_1, f_2) \implies \mathit{loc}_m(s, b_2, f_2) \end{aligned} \qquad (\mathrm{CHC}_{FM})$$

where s is the same as in CHC_A; each b_i is a subset of their respective a_i in CHC_A (the cells encoded using arrays); b_0' and b_1' are, respectively, subsets of b_0 and b_1, for the cells encoded using arrays in the function call fun; f_i are vectors of finite maps representing a subset of a_i; and f_{in}, f_{out} finite maps used as parameters in the function call. The constraint $\phi_A(b_0, f_{out}, b_1)$ describes how the values in the output finite maps f_{out} are related to the arrays b_1 in the caller. Such constraints are generated if a memory cell is inferred to be unbounded in the

```
EncFunCall($g$, alias, $params$)
  $APs :=$ compFiniteAPs($g$, alias, $params$)
  $\phi := true$
  $sorts :=$ infer-sorts($APs$, alias)
  for all $(var, ap) \in APs$ do
    match $ap$ with cons($c, ap'$) →
      $k :=$ encodeAP($ap', var, sorts$)
      $v :=$ encodeAP($ap, var, sorts$)
      $c_r =$ alias-rep(alias, $c$)
      $Ps_{in}[c_r] := Ps_{in}[c_r] \cup \{(in(k), in(v)\}$
      $Ps_{out}[c_r] := Ps_{out}[c_r] \cup \{(out(k), \texttt{mk-var}())\}$
  for all $(c_r, Ps) \in Ps_{in}$ do
    $Args_{in}[c_r] := \texttt{mk-fm}(Ps)$
  for all $(c_r, Ps) \in Ps_{out}$ do
    $Args_{out}[c_r] := \texttt{mk-fm}(Ps)$
    $\phi' := in(cellToE(c_r, sorts[c_r]))$
    for all $(k, v) \in Ps_{out}$ do
      $\phi' := \texttt{mk-write}(\phi', k, v)$
    $\phi := \phi \wedge \texttt{mk-eq}(out(cellToE(c_r, sorts[c_r])), \phi')$
  return $(Args_{in}, Args_{out}, \phi)$

encodeAP($ap, var, sorts$)
  match $ap$ with
    nil → return $varToE(var)$
    cons($(n, o), ap'$) →
      $MS := cellToE((n, o), sorts[(n, o)])$
      $idx' :=$ encodeAP($ap', var, sorts$)
      $idx := \texttt{mk-add}(idx', o)$
      return $\texttt{mk-read}(MS, k)$

mk-read($mem, k$)
  match sort($mem$) with
    Array → return $mem[k]$
    FiniteMap → return get($mem, k$)

mk-write($mem, k, v$)
  match sort($mem$) with
    Array → return $mem[k \leftarrow v]$
    FiniteMap → return set($mem, k, v$)
```

Fig. 6: Algorithm to encode the finite memory at a function call.

caller and bounded in the callee. The constraint $\phi_{FM}(f_0, f_{out}, f_1)$ describes how the values in the output finite maps are related to the finite maps f_1 in the caller. Such constraints are generated if a memory cell is inferred to be bounded both in the caller and in the callee but the caller may access more memory locations than the callee, and thus they have a different size.

Extending the encoding. We present the parts of the CHC encoding related to memory. Memory accesses are modeled either with arrays or finite maps. The function $cellToE(c, sort)$ takes a memory cell c and its $sort$ and returns a logical variable of that sort. The sort is determined by the algorithm described in Fig. 3 (Sec. 4). If c is finitely accessed, its sort is a finite map of size the number of access paths to it, otherwise, it is an array.

Without function calls, for every memory operation, its associated memory cell c is obtained from the pointer analysis. Then, $cellToE$ is used to encode c as an array or finite map. The remaining operands are encoded by the function $varToE$, which takes a program variable and returns a logical variable (pointers are encoded as integers). The functions `mk-read` and `mk-write`, defined in Fig. 6, produce the array or finite map term for the corresponding memory operation.

Function calls require additional constraints. Namely, the formulas ϕ_A and ϕ_{FM} in CHC$_{FM}$, and the finite maps that represent the memory used by the function. Fig. 6 shows how to encode a function call. EncFunCall takes as input the graph g of the called function, and the aliasing (`alias`) and the parameters ($params$) at the call site. It returns a triple $(Args_{in}, Args_{out}, \phi)$, with $Args_{in}, Args_{out}$ mappings from equivalence classes of cells to the corresponding finite map used to encode all the cells in the class (i.e., respectively, f_{in} and f_{out} in CHC$_{FM}$), and ϕ that expresses $\phi_A \wedge \phi_{FM}$ in CHC$_{FM}$. For simplicity, $Args_{in}$ and $Args_{out}$ are defined only for cells belonging to finitely accessed equivalence classes. The remaining cells are encoded as arrays.

Functions of the form `mk-E` build a logical expression of sort `E`. The functions `mk-eq` and `mk-add` are self-explanatory. `mk-var` returns a fresh integer variable. `mk-fm` builds a finite map out of a set of pairs of key-values. The function alias-rep(`alias`, c) returns the representative of the class of c induced by `alias`.

The algorithm proceeds as follows. First, all the access paths are computed on line 1 (described in Fig. 3). Based on these, on line 3, the sorts of the finite maps are inferred. The loop on lines 4-10 processes all access paths. On lines 6-7, the sequence of dereferences corresponding to the access path is encoded as key-value pair of logical expression. The value is the whole sequence and the key is the sequence except the last dereference. The algorithm produces *input* and *output* finite maps representing memory before and after the call (lines 12-14). The functions *in* and *out* rename logical terms on the set of input and output variables. Finally, lines 15-18 build ϕ_A and ϕ_{FM} described in CHC$_{FM}$.

In function encodeAP, if the access path (AP) is empty, the logical expression of the pointer *var* is returned using *varToE*. If not, first, the formula of the rest of the AP is computed, which is the index of the current level of the AP. *MS* is the logical expression for the cell of the current level of the AP. For example, if *cellToE* maps cells $(N3, f0)$, $(N2, f0)$, and $(N1, f0)$ respectively to a_3, a_2, and a_1, an expression of the form `p1->f0->f0->f0` with the access path $(p1, [(N3, f0), (N2, f0), (N1, f0)])$ of Ex. 3 is encoded as: $a_3[a_2[a_1[p1]]]$

The program defined by Figs. 1a and 1c encoded with finite maps is:

$$v = \texttt{get}([s \mapsto v_{in}], s) \rightarrow \mathit{read\_x}(s, v, [s \mapsto v_{in}]) \tag{CHC 5}$$

$$\mathit{true} \rightarrow \mathit{init\_x}(s, [s \mapsto v_{in}], \texttt{set}([s \mapsto v_{in}], s, 0)) \tag{CHC 6}$$

$$\begin{aligned} B3a \wedge \mathit{init\_x}(p, [p \mapsto m_2[p]], [p \mapsto v_{out}]) \wedge & \qquad \text{(L7a)} \\ m_3 = m_2[p \leftarrow \texttt{get}([p \mapsto v_{out}], p)] \ \wedge \mathit{read\_x}(q, r, [q \mapsto m_3[q]]) \rightarrow r = 20 & \qquad \text{(CHC 7)} \end{aligned}$$

Up to literal L7a, the same constraints as in CHC 3 are produced. The arguments in L7a are generated on lines 12 and 14 of the algorithm. The last line of CHC 7 captures how the output finite map and the memory at the call m_3 are related (lines 15-18). After applying the rules in Sec. 5 to remove finite map expressions we obtain:

$$v = v_{in} \rightarrow \mathit{read\_x}(s, v, s, v_{in}) \tag{CHC 5 without finite maps}$$

$$\mathit{true} \rightarrow \mathit{init\_x}(s, s, v_{in}, s, 0) \tag{CHC 6 without finite maps}$$

$$B3a \wedge \mathit{init\_x}(p, p, m_2[p], p, v_{out}) \wedge \tag{L7a without finite maps}$$

$$m_3 = m_2[p \leftarrow v_{out}] \wedge \mathit{read\_x}(q, r, q, m_3[q]) \rightarrow r = 20 \tag{CHC 7 without finite maps}$$

7 Experimental Evaluation

We have implemented our new technique to encode bounded memory regions as finite maps using the CHC-based model-checker SeaHorn. The implementation is available in `https://github.com/seahorn/seahorn/releases/tag/fmaps-sas22`. We have evaluated it on two different sets of benchmarks.

	SeaHorn		UAuto	
	mod	fmap-mod		
	time (s)	time (s)	time (s)	quantified
bench1	1	1	8	Yes
bench2	–	1	20	Yes
bench3	–	1	18	Yes
bench4	–	1	120	Yes
bench5	–	1	–	–
bench6	–	8	–	–

Table 1: SeaHorn (mod, fmap-mod), and UAuto on micro-benchmarks.

Evaluation on microbenchmarks. To evaluate our technique we handcrafted a set of benchmark problems.[6] This is a set of small but challenging benchmarks for modular, SMT-based model-checking. These examples can be easily verified by inlining the functions, however, as we can see later, inlining does not scale for larger programs. This means that if any of the patterns in these examples are present in some program, it will not be possible to verify it when inlining is not feasible. Table 1 shows the result of our evaluation. We compare SeaHorn with two different modular encodings: modeling memory only with arrays (mod) and our proposed technique, modeling memory with arrays and finite maps (fmap-mod). SeaHorn, regardless the encoding, can only produce quantifier-free summaries. As a result, it diverges in the cases where only quantified summaries exist. We also compare with UAutomizer [13] (UAuto), which can also produce (quantified) function summaries[7]. Table 1 shows whether the summaries discovered by UAuto are quantified. The symbol '–' denotes that tool did not produce an answer in 5 minutes.

Evaluation on SV-COMP programs. We have also evaluated our approach on a selection of 745 Linux device drivers from SVCOMP 2019[8], after discarding all the benchmarks that were trivially proven by the SeaHorn front-end or produced some crash. These programs are large and use a variety of language features including pointers and aliasing. All experiments were run on Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz with 48 cores and 251GB of RAM on Ubuntu 18.04.

Although SeaHorn is actively maintained, it does not participate in SV-COMP. Hence, we first compare SeaHorn with participants of SVCOMP 2021 which also focus on discovering safe inductive invariants. Table 2 shows a comparison with UAutomizer [13] (UAuto) and CPAChecker [5] (CPA). We compare

[6] Available at `https://zenodo.org/record/4505518`.

[7] In this evaluation, we used the online version of UAuto `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=int&tool=automizer` because it is the one that computes function summaries.

[8] Available at `https://zenodo.org/record/4498784`.

	UAuto	CPA	SeaHorn (mono)
false	2	17	41
true	94	226	218

Table 2: Instances solved out of 745 within 900s and 8GB of memory.

with their most recent versions[9], customized to analyze Linux device drivers. For SeaHorn, we use monolithic encoding using arrays to model memory. The rows true and false show how many instances were proven and disproven (i.e., the property holds or is violated), respectively, without exhausting resources. In the rest, solved instances are those for which the verifier produced an answer. From this comparison, we can safely conclude that SeaHorn is competitive with UAuto and CPA on our benchmarks.

Tables 3-5 show the main results of this paper by comparing our new encoding (fmap-mod) with two baseline encodings already available in SeaHorn: one monolithic encoding with multiple arrays (mono) where all functions have been inlined[10] and one modular encoding with multiple arrays (mod) without special treatment of statically-known finite memory. Since we are more interested in the comparison with mod, the column mod shows the best result after 5 runs on each program.

During our evaluation, we found out that representing all finite memory with finite maps can be expensive. We hypothesize that the correctness of some Linux device drivers does not depend much on memory (especially after the optimizations performed by the SeaHorn frontend). In those cases, the solver can avoid reasoning about most of the array expressions. However, our encoding with finite maps eagerly adds constraints about memory, regardless of whether they are relevant to prove the program correct or not.

For this reason, we limit the size of the finite maps (the number of key-value pairs), denoted by sX in Tables 3-5, where each finite map of size X is encoded using $2X$ scalar variables, two per key-value pair. Moreover, when no relations about the keys are known, all cases need to be considered. In the worst case, for a finite map of size Y, an *ite* term of depth $Y - 1$ is created for *get* operations, and Y *ite* terms of depth $Y - 1$ (one per key-value pair) are needed in predicate calls (see the reduction rules in Sec. 5). Therefore, we also limit this, denoted by aY in Tables 3-5, informally meaning "encoding with finite maps only the memory objects pointed by at most Y pointer variables in the program". In these tables, the column best is equivalent to running in parallel all finite map configurations in a portfolio and stopping when the first one is solved. This is more resource intensive than other configurations. However, since the optimal

[9] Available at `https://github.com/ultimate-pa/ultimate/releases/tag/v0.2.1` and `https://cpachecker.sosy-lab.org/CPAchecker-2.0-unix.zip`, respectively.

[10] Recursive functions are not relevant to prove the properties so that they are abstracted by functions without side-effects that return non-deterministic values.

	mono	mod	fmap-mod										
			s1-a1	s2-a1	s2-a2	s3-a1	s3-a2	s3-a3	s5-a1	s5-a2	s5-a3	s5-a5	best
false	41	107	94	91	93	89	91	90	90	90	90	87	110
true	218	278	265	268	262	270	263	263	265	256	261	262	297
total	**259**	**385**	359	359	355	359	354	353	355	346	351	349	**405**

Table 3: Instances solved by SeaHorn encoding as mono, mod, and fmap-mod.

	s1-a1	s2-a1	s2-a2	s3-a1	s3-a2	s3-a3	s5-a1	s5-a2	s5-a3	s5-a5	best
false	61	59	59	56	59	58	58	59	59	57	73
true	85	91	82	89	82	84	88	79	82	83	110
total	146	150	141	145	141	142	146	138	141	140	183

Table 4: Instances solved by SeaHorn with fmap-mod not solved by mono.

	s1-a1	s2-a1	s2-a2	s3-a1	s3-a2	s3-a3	s5-a1	s5-a2	s5-a3	s5-a5	best
false	2	1	2	2	4	1	1	1	2	1	5
true	6	11	11	8	11	13	8	15	13	14	24
total	8	12	13	10	15	14	9	16	15	15	29

Table 5: Instances solved by SeaHorn with fmap-mod not solved by mod.

	s1-a1	s2-a1	s2-a2	s3-a1	s3-a2	s3-a3	s5-a1	s5-a2	s5-a3	s5-a5	best
false	15	17	16	18	20	18	18	18	19	21	2
true	19	21	27	16	26	28	21	37	30	30	5
total	34	38	43	36	46	46	39	55	49	51	7

Table 6: Instances solved by SeaHorn with mod (best out of 5 runs) not solved by any configuration of fmap-mod.

finite map configuration for each program cannot be known a priori, it is a best effort to verify as many programs as possible.

Table 3 contains the number of solved instances per encoding (columns). The row **total** is the number of benchmarks solved by each configuration with the available resources. Tables 4 and 5 show how many instances the CHCs with finite maps (fmap-mod) were solved that, respectively, for mono and mod it was not possible to solve, split by false and true. For example, in Table 5, s3-a2 (finite maps of size 3 and at most 2 keys may alias) solves 4 false and 11 true instances that cannot be solved by mod. The best configuration of finite maps proves 183 benchmarks that mono could not, and 29 that mod could not. However, fmap-mod could not solve all the instances that mod solved. Table 6 shows the number of instances that were solved only by (the best out of five runs of) mod and not by any fmap-mod configuration (one run), represented in each of the columns. There were 7 instances proven by mod that no fmap-mod configuration proved (shown in the best column of Table 6). Lastly, were 35 mono instances that no mod or fmap-mod configuration proved.

We found that out of all the true instances solved by mod, 23% required arrays in the summaries. When encoding memory with fmap-mod configurations, only 9% of the summaries required arrays on average.

Finally, we do not report the time of the encoding phase because it is negligible compared with the time spent solving. SeaHorn already performs a whole-program pointer analysis so the overhead of our new encoding (Sec. 6) and the finite maps reduction (Sec. 5) is very low.

8 Conclusions

We presented a new CHC encoding that enables automatic modular proofs for programs with pointers without using quantified summaries. The main idea is to encode explicitly the finite parts of the frame of a function when they can be statically determined. We presented an algorithm to infer statically the size of the memory used by a function. To represent bounded memory succinctly, we proposed a new theory of finite maps, adapted to CHCs, and a reduction procedure to simpler theories supported by any SMT solver. We then extended a CHC encoding to represent finite memory using finite maps. We implemented our new technique in SeaHorn and evaluated it on Linux device drivers. Our results are encouraging and show that our new encoding can prove new programs that a previous encoding cannot. However, our evaluation also shows that a priori knowledge about the program and its properties can help to choose the most effective encoding of CHCs. We consider this problem an interesting future work.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: An interpolation-based algorithm for inter-procedural verification. In: VMCAI. pp. 39–55 (2012)
2. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: CAV. pp. 672–678 (2012)
3. Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 50–62. ACM (2020)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT **9**(5-6), 505–525 (2007)
5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: CAV. pp. 184–190 (2011)
6. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. pp. 24–51 (2015)
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Shao, Z., Pierce, B.C. (eds.) POPL. pp. 289–300 (2009)

8. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. J. Autom. Reason. **64**(7), 1393–1418 (2020)
9. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. pp. 168–176 (2004)
10. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. pp. 343–361 (2015)
11. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: SAS. pp. 148–168 (2017)
12. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: ATVA. pp. 248–266 (2018)
13. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol - (Competition Contribution). In: TACAS. pp. 641–643 (2013)
14. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 471–482. ACM (2010)
15. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4085, pp. 268–283. Springer (2006). https://doi.org/10.1007/11813040_19, https://doi.org/10.1007/11813040_19
16. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV. pp. 17–34 (2014)
17. Kröning, D., Weissenbacher, G.: A proposal for a theory of finite sets, lists, and maps for the smt-lib standard (2009 (accessed October 13th, 2020)), http://www.philipp.ruemmer.org/publications/smt-lsm.pdf
18. Kuderski, J., Navas, J.A., Gurfinkel, A.: Unification-based pointer analysis without oversharing. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019. pp. 37–45. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894275, https://doi.org/10.23919/FMCAD.2019.8894275
19. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, https://doi.org/10.1007/978-3-642-17511-4_20
20. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969), reprinted in McC90
21. Merz, F., Falke, S., Sinz, C.: LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: VSTTE. pp. 146–161 (2012)
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning . In: Pretschner et al. [24], pp. 104–125. https://doi.org/10.3233/978-1-61499-810-5-104, https://doi.org/10.3233/978-1-61499-810-5-104

23. Piskac, R., Wies, T., Zufferey, D.: Grasshopper - complete heap verification with mixed specifications. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8413, pp. 124–139. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_9, https://doi.org/10.1007/978-3-642-54862-8_9
24. Pretschner, A., Peled, D., Hutzelmann, T. (eds.): Dependable Software Systems Engineering. Vol. 50 of Pretschner et al. [24] (2017), http://ebooks.iospress.nl/volume/dependable-software-systems-engineering-3
25. Rakamaric, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: CAV. pp. 106–113 (2014)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74 (2002)
27. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5653, pp. 148–172. Springer (2009). https://doi.org/10.1007/978-3-642-03013-0_8, https://doi.org/10.1007/978-3-642-03013-0_8
28. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. pp. 32–41 (1996)
29. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 29–37. IEEE Computer Society (2001). https://doi.org/10.1109/LICS.2001.932480, https://doi.org/10.1109/LICS.2001.932480