Extension of the ELDARICA C model checker with heap memory

Zafer Esen
Abstract

**Extension of the ELDARICA C model checker with heap memory**

Zafer Esen

Model checking is a verification method which is used to detect bugs which would be extremely hard to detect using traditional testing, and ELDARICA is a state-of-the-art model checker which accepts a variety of formats as its input, including programs written in a fragment of the C language. This thesis aims to improve the C front-end of ELDARICA to a point where it can automatically model and verify C programs which contain pointers, heap memory interactions and structs, which are currently not supported.

This work models the heap in a similar way to how it was done in JayHorn, a model checker for Java, by automatically finding quantified invariants which summarize the states of data structures that are on the heap. Support for structs is added by modeling them as algebraic data types, and limited support for stack pointers is added with some constraints on how they are declared and used.

The initial experimental results are promising. The extended tool can now parse programs written in a larger fragment of the C language, with acceptable precision and performance in comparison to similar tools.
Acknowledgements

I am deeply grateful to my supervisor Philipp Rümmer for his guidance, his patience during many hours of discussions, and for always being there even when he was extremely busy.

I also want to thank Mohamed Faouzi Atig for reviewing my work and for providing valuable insight into this report.

Finally, I would like to thank my parents and my family for their continuous support; and my wife Iuliia, for standing by me and providing care and encouragement during my education.
List of Figures

1.1 Eldarica Architecture ........................................... 14
1.2 TriCera Architecture ............................................ 15
1.3 CIMP and TriCera workflow ...................................... 16

2.1 Verification using Horn clauses ................................. 17
2.2 A simple C program and its program graph .................. 19
2.3 Horn encoding of the program given in Figure 2.2 .......... 20
2.4 Another simple C program and its CHC translation ........ 20
2.5 Space invariants in JayHorn .................................... 21
2.6 Separation logic picture semantics [30] ...................... 22

3.1 The abstract syntax of CIMP ..................................... 24

6.1 toHorn for basic programs ...................................... 38
6.2 exp function ....................................................... 39

7.1 A nested struct as a tree ........................................ 43

8.1 push and pull operations ......................................... 46
8.2 Representing the heap using heap invariants ............... 47
8.3 toHorn extension for heap ..................................... 50

9.1 Type hierarchy in TriCera ....................................... 51
# List of Tables

4.1 Conversion of syntactic sugar syntax ...................................... 28
4.2 Stack vs heap pointers ......................................................... 30
4.3 Simplification examples ....................................................... 32

5.1 Rule for regular assignment .................................................. 34
5.2 Branching (if-else) and looping (while) statements ................... 34
5.3 load, store, malloc and calloc .............................................. 35
5.4 assert & assume statements .................................................. 35

10.1 Results - 1 minute timeout ................................................. 58
10.2 Results - 5 minutes timeout ................................................. 58
Chapter 1

Introduction

Computer programs today are complex structures which might contain millions of lines of code. They are used in a broad range of industries ranging from casual entertainment to safety critical ones such as autonomous driving, health care and defense. Software programs in many industries are in the trend of getting larger and more complex. In 2006, just 30 years after the first software was ever used in a car, it was estimated that 20 to 28% of the production costs of a car were due to software development [1].

Unfortunately, any part of software also has a high probability of containing bugs. Complex and large programs lead to more bugs which might go undetected, and in safety-critical systems these bugs might lead to very costly failures [2]. There are a few ways to detect and eliminate these bugs in order to increase software quality.

Testing is one such method for checking computer programs dynamically, with the intent of reducing the number of bugs. It is dynamic, because the tests are carried out by executing the software. Testing is widely used in the industry; however, its success depends on the experience of the people writing the test cases and executing the tests, as the automated tools are still far from satisfactory [3]. Various forms of coverage are frequently used to determine whether a test is sufficient; however, even with full coverage and extensive test cases, testing might still miss some bugs. Although it is always useful to do testing, as famously stated by Dijkstra, "testing can only find bugs, not prove their absence".

Model checking is a method for automatically checking whether a given system (or software) meets its specification. A model of the actual system is automatically created, then a mathematical proof is attempted in order to show that the model conforms to its specification. If it does not, then the model checker shows how the bug can be reproduced (with a counterexample trace).

Like testing, model checking has its limitations; however, its main strength is to find bugs which would be extremely hard to find and reproduce using traditional testing. As a result, it is widely used for the verification of hardware and software in the industry, ranging from tools which verify safety-critical space and aircraft software [4] to tools such as the Facebook Infer, which is used to speed up the software development cycle and reduce costs in a fast paced industry [5].

ELDARICA is one such tool for model checking. It is a state-of-the-art solver for Horn clauses, which accepts a variety of formats as its input, with C programs being one of them. The input programs are automatically translated into Horn clauses (i.e. the model). It uses Predicate Abstraction with Counterexample-Guided Abstraction Refinement (CEGAR) to check whether these Horn clauses are satisfiable, and provides a counterexample trace if a solution cannot be found [6]. Figure 1.1 shows the main architectural components of
ELDARICA.

The main goal of this paper is to extend the Horn encoder of ELDARICA, which can parse a subset of programs written in C. This subset currently excludes support for pointers, arrays, structs and heap memory, to name a few.

1.1 Aim

The goals of this thesis are to:

- add support for modeling structs as Horn clauses,
- find a method for encoding heap memory using Horn clauses,
- expand the Horn encoder of ELDARICA by implementing support for heap memory and pointers (excluding pointer arithmetic),
- evaluate the performance of the implementation in comparison to other C model checkers; such as SMACK\(^1\), CPAchecker\(^2\) and SeaHorn\(^3\).

The overall goal can be stated as improving ELDARICA to a point where it can automatically model and verify C programs which contain pointers, heap memory interactions and structs.

1.2 Definitions and Acronyms

1.2.1 TriCera

While the thesis was going on, ELDARICA’s Horn encoder for C programs was separated from the main software, and given the name TriCera\(^4\). Since the main goal of this thesis is to extend ELDARICA’s Horn encoder for C programs, this separation means that the goal is updated as extending TriCera.

TriCera still uses ELDARICA as the backend to solve the generated Horn clauses; however, this separation of concerns (i.e. generating the Horn clauses and solving them) means that the backend can be easily switched from ELDARICA to another Horn clause solver if desired.

A diagram depicting the TriCera architecture is given in Figure 1.2.

\(^{1}\)http://smackers.github.io/
\(^{2}\)https://cpachecker.sosy-lab.org/
\(^{3}\)http://seahorn.github.io/
\(^{4}\)https://github.com/uuverifiers/tricera
1.2.2 CIMP Language

TriCera can parse a subset of C programs which excludes arrays and pointer arithmetic; however, with some non-C additions like networks of timed automata with unbounded parallelism support, clocks, binary communication channels, and time invariants [7].

For the sake of reducing complexity, this report considers a subset of this language which TriCera can parse. Some simplifications are then carried out on this language to simplify parsing, which are explained in detail in Chapter 4. The end result of this simplification is an intermediate language, which is called CIMP. The relation of CIMP to C and the TriCera input language is shown in Figure 1.3(a).

The idea of simplifying input programs is not novel in verification. VeriMap, another tool which generates Horn clauses from C programs, first simplifies programs to the C Intermediate Language (CIL) [8]. TriCera does this simplification and translation into Horn clauses in a single step, so CIMP is a language defined just for the purpose of explaining the work done in this thesis, and it is not an actual part of TriCera.

Figure 1.3(b) shows the workflow of the TriCera Horn encoder. Input C programs are first internally reduced to CIMP programs, and then the Horn clauses are generated using this intermediate representation. Note that this intermediate representation is not exposed to the outside world.

1.2.3 The Stack and The Heap

This document refers to the non-dynamic memory where local and global variables reside, as the stack. The local and global variables collectively form the stack variables. Pointers pointing to the stack variables are called stack pointers.

On the contrary, the dynamic memory allocated using the C functions malloc and calloc is called the heap. Pointers which point to the heap are called heap pointers.

1.2.4 Other Definitions and Acronyms

ADT: Algebraic Data Type
CHC: Constrained Horn Clause
SOS: Structural Operational Semantics
TS: Transition System
1.2. DEFINITIONS AND ACRONYMS

CHAPTER 1. INTRODUCTION

(a) TriCera accepts inputs written in a subset of C with some additions such as timed automata and clocks. The diagram depicts this relation (blue and yellow boxes).

Since the focus of this thesis is extending TriCera by adding support for pointers (which point to the stack or the heap) and C structs, for clarity, only a subset of the TriCera input language is considered with the additions of pointers and structs. In this figure, this subset is named TriCeraS. Chapter 4 introduces load & store operations in order to deal with these inputs. A toy language named CIMP is defined in Chapter 4 to TriCeraS. It introduces load & store operations after applying the simplification rules described in Chapter 4 to TriCera. As inputs are mappings of CIMP to C, load & store operations are mapped to the heap or to the stack. To simplify things written in C, a number of syntactic sugar expressions and statements such as

Improve written in C might be described using this subset actually needed to TriCera too.

(b) TriCera Horn encoder workflow

Figure 1.3: (a) Relation of CIMP to C, (b) TriCera workflow

TriCera accepts inputs written in a subset of C with some additions such as timed automata and clocks. The diagram depicts this relation (blue and yellow boxes).
Chapter 2

Background

This chapter first gives some background regarding Horn clauses and constrained Horn clauses (CHCs). Then the challenge of modeling the heap (the main goal of this thesis) and the various approaches in the literature to overcome this problem are discussed.

2.1 Constrained Horn Clauses for Program Verification

2.1.1 Overview

The significance of Horn clauses was first proposed by [9]; and the use of Horn clauses for program verification was first proposed in [10]. The popularity of using Horn clause solving as a uniform framework for program verification has been increasing since then, and Horn clause based program verification has been the subject of many research papers [11]–[15]. Eldarica [6], JayHorn [16], SeaHorn [17] and Z3 [18] are some of the tools which utilize Horn clause solving.

The main idea is to convert programs and specifications into a set of constrained Horn clauses, and use an off-the-shelf Horn clause solver, such as Eldarica or Spacer [19], to prove that no error states are reachable (i.e., the program is correct). If the clauses are unsolvable (i.e., the program is incorrect), then a counterexample is provided in order to expose program errors. A diagram depicting the idea of verifying with Horn clauses is given in Figure 2.1. The orange box represents the off-the-shelf Horn clause solver.

![Figure 2.1: Verification using Horn clauses](image-url)
2.1. HORN CLAUSES

2.1.2 Horn Clauses

Some definitions in predicate (first-order) logic must be given before defining a Horn clause in the same logic:

- A **term** is defined as a constant, a variable or an application of a function to a term (e.g. $x$, 3, $x + 5$),

- A **literal** is defined as any predicate or its negation applied to first-order terms (e.g. $Left(x)$, $\neg Left(y)$),

- A **clause** is defined as a disjunction of literals where the variables are universally quantified (e.g. $\forall x : Left(x) \lor Right(x)$).

- An **atomic formula** or **atom** is a formula of the form $P(t_1, ..., t_n)$ where $P$ is a predicate applied to the terms $t_n$.

A Horn clause is defined as a clause that contains at most one positive literal, e.g.

$$l \lor \neg l_1 \lor ... \lor \neg l_n$$

Horn clauses in the form given above are called **definite clauses**, and programs defined by these clauses are called **definite programs**. A Horn clause with no negative literals is called a **fact**. Horn clauses form the basis of logic programming. E.g. the programming language Prolog is based on Horn clauses.

In logic programming notation, Horn clauses are usually written in the implication form $l \leftarrow l_1 \land ... \land l_n$ or in Prolog notation as $l : - l_1, ..., l_n$.

2.1.3 Constrained Horn Clauses

A constrained Horn clause (CHC) in predicate logic is a formula

$$H \leftarrow \neg C \land B_1 \land ... \land B_n$$

either an application $p(t_1, ..., t_k)$ of a k-ary predicate to first-order terms or false

As shown in the formula, if the constraint theory for $C$ is linear arithmetic, the constraints can be expressed using the relation symbols $<$, $\leq$, $>$, $\geq$ and $=$.

The terms in the clauses represent the program variables and other variables which are introduced while translating the input program into CHCs.

Solving a set of Horn clauses requires assigning a formula to each predicate, such that their first order interpretation makes all the clauses **true**. If the set of Horn clauses is not solvable, then the proof of refutation gives the counterexample trace.

2.1.4 Transition Systems and Program Graphs

In order to give the necessary intuition as to how Horn clauses can define programs and be used in verification, the definition of a transition systems (TS) will be first given. A TS can be defined as the tuple $(S, I, \rightarrow)$, where

- $S$ is the state space;

- $I \subseteq S$ is the set of initial states;
Software programs can be represented as transition systems by defining the states as the Cartesian product of program control locations (Loc) and the valuation of the variables (Val) at those locations, $S = Loc \times Val$. Then the initial states can be defined as the Cartesian product of the set of initial locations and the set of initial variable valuations, $I = Loc_{init} \times Val_{init}$.

To prove the safety of transition systems, a set representing the error states is first introduced, $Err \subseteq S$. Then, a transition system is said to be safe if there is no path which touches an error state; i.e. there is no path $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$ with $s_0 \in I$ and $s_n \in Err$.

Adding conditions to the transitions results in a system which is called a Program Graph (or a control flow graph) [20]. The variable valuations are also replaced with effects on transitions. Program graphs can be translated into transition systems via unfolding.

For the example program given in Figure 2.2(a), $Loc = \{l1, l2, l3, l4, l5, l6\}$, $Err = \{err\}$, $Init = \{l1\}$. The two variables x and y are integers, so they can have valuations which are integer values. Assuming initially $x$ and $y$ have the values 0, $Val_{init} = \{(0,0)\}$, the entry point of the program becomes $I = (l1, 0, 0)$. The program graph is given pictorially in Figure 2.2(b). This system can be expressed as a program graph as shown in Figure 2.3 on the left, and in CHC form on the right.

As can be seen in Figure 2.3, it is relatively straightforward to encode basic programs as logic clauses, and the clauses strongly resemble the program graph. The effects are applied to the predicates, and the conditional transitions are written as constraints in conjunction to the body of the clauses.

In the example Horn encoding, the entry point is represented with the body being empty, and the error state is represented by the head being false. The defined predicates all have an arity of two, as the two program variables are $x$ and $y$. The program is safe if the error state can never be reached, which is true only if the clauses are unsatisfiable.

Another example translation into CHCs for a code containing a loop is given in Figure 2.4.
2.2. The Challenge

The main challenge of modeling heap memory stems from the fact that heap memory is dynamically allocated, and a static model of it is hard to find. While data on the stack is bounded, a heap allocated data structure might grow unbounded during the execution of a program. Also, invariants about the heap are usually very complicated.

In software model checking and most other formal methods, instead of direct reasoning using the operational semantics of programs, an approximate model of the semantics is often used [21]. This is called abstraction, and abstractions are commonly used while modeling the heap as well [17], [22]–[24]. The abstractions can range from mapping the whole heap to a single abstract object to having separate heap mappings depending on allocation sites and data types (much more precise).

Having non-precise abstract objects reduces the analysis time. It is not possible to achieve full precision (i.e., completeness - no false positives) while keeping the soundness (i.e., no false negatives) and termination properties of the analysis. It is proven to be undecidable to even statically analyze if two pointers point to the same location (known as alias analysis) for dynamically allocated objects, which would be necessary to achieve completeness while statically modeling the heap [25].

Unlike Java which does not even support stack pointers, most of the pointers in C programs are actually stack pointers [26]. This means that even a very high level of heap abstraction is usually sufficient to achieve good verification results for most C programs.

Modeling the heap with more than basic precision also requires the use of pointer analysis. This is required in order to distinguish if two pointers alias, and partition the heap into finer grained regions in order to increase precision if they do not.
2.3 Related Work

SeaHorn implements heap as a collection of non-overlapping arrays [17]. The level of abstraction while modeling statements is adjustable, and the heap is modeled only when the finest level of abstraction is chosen. This explicit modeling is achieved by utilizing a variant of the pointer analysis method called Data Structure Analysis (DSA) [27].

SMACK also utilizes DSA in order to partition the memory into non-overlapping arrays [22].

CBMC provides support for heap memory as well, however, the analysis is bounded. It checks if there are any memory leaks (i.e. allocated memory is not freed before program termination), and whether the accessed or freed pointer still points to a object (i.e. null pointer dereferencing) [23].

Jayhorn introduces the concept of space invariants, which are used to automatically abstract the heap interactions [24]. The main idea is that instead of modeling each Heap location precisely, the invariant models the properties that hold for the heap at each program location. A simple pictorial description of the space invariants is given in Figure 2.5. Refinements are done in order to increase the precision of the model, such as adding flow sensitivity to the invariants and inlining the methods.

![Figure 2.5: Space invariants in JayHorn](image)

Separation logic Separation logic, which is an extension of Hoare logic, also allows reasoning about the structure of heap memory [28]. It solves the failure of the frame rule in Hoare logic, which is caused by aliases, by introducing the separating conjunction $\ast$ which reads as “and separately in memory”. This enables reasoning about programs by expressing their logic with only in-place updates of memory (i.e. no aliases). Figure 2.6 pictorially shows the main idea of separation logic. $x$ and $y$ are both pointers, separately in memory, pointing to each other. Thus, the heap can be decomposed into two separate parts (called heaplets by the authors). The downside is, it is not easy to automate separation logic as it is very expressive and usually the tools employing it are restricted to only work with the decidable fragments [29].
2.4 Approach

TriCera models heap using a method similar to what was done in JayHorn [16]; although currently at a higher abstraction level which lacks the refinements to increase precision. The other C model checkers discussed in Section 2.3 use different methods to model the heap; so the method used here can be considered novel for a C model checker. The details of the heap model will be given in Chapter 8.

Most of the other contributions of this thesis are about extending the capabilities of TriCera, so a wider range of C programs can be verified. This includes basic support for stack pointers, support for C structs and basic support for heap memory. The given semantic rules and translation into Horn clauses, although they are for a subset of the whole TriCera input language, are also provided for the first time.

Figure 2.6: Separation logic picture semantics [30]
Chapter 3

Syntax of the Simplified Input Language (CIMP)

The syntax of CIMP is given in Figure 3.1. CIMP is obtained after the simplifications explained in Chapter 4 are done on an accepted subset of the language (see Figure 1.3). For example, TriCera support pointers and assignments to struct fields; however, these are not in CIMP syntax as they are replaced by other statements or expressions during simplification.

The two main nonterminals are Statement for statements and Expr for expressions. Expressions of CIMP are side-effect-free.

Note that an assumption is made that all variables are already declared and assigned a Type for the sake of simplicity, so the syntax does not cover initialized or uninitialized variable declarations.

The address-of operator (&) is also not in the syntax. For stack pointers it is only possible to use this operator during initialization due to the limitations that will be described in Chapter 4.3, and since the syntax does not cover variable declarations, the operator is not shown in the table. For heap pointers, although the use of the address-of operator is supported in TriCera, it is omitted in this report for simplicity. This is because heap pointers are created via memory allocation functions, and the use of the & operator is unnecessary in most cases.

In the rest of this document, e is used as shorthand for Expr, and S is used as shorthand for Statement. The terminals and non-terminals in the syntax can also appear subscripted (e.g. e₁, S₂, xᵢ etc.).
Program ::= Statement

Statement ::= Statement , Statement
| x = Expr                       assignment
| x = malloc(Type)               uninitialized heap allocation
| x = calloc(Type)               zero-initialized heap allocation
| x = load(Expr)                load operation
| if(Expr) {Statement} else {Statement} conditional statement
| while(Expr) {Statement}        while loop
| store(Expr, Expr)             store operation
| assert(Expr) | assume(Expr) assertion and assumption
| skip                         no operation

Expr ::= x                       a variable
| v                             a value
| Expr.f                        field access
| UnOp Expr                     unary operation
| Expr BinOp Expr               binary operation
| (Type) {fi = vi i∈1..n}        compound literal

UnOp ::= - | !    arithmetic operator

BinOp ::= + | - | * | / | %    relational operator
| < | <= | > | >= | == | !=    logical operator
| && | ||

Type ::= int    integer type
| struct {fi : Ti i∈1..n}    struct type with n fields
| Type*                  pointer to type

v ::= x                       a variable reference
| n                          integer value
| {fi → vi i∈1..n}            a struct value
| l                          heap location

n ::= ... | -1 | 0 | 1 | ...

Figure 3.1: The abstract syntax of CIMP.

The grammar of CIMP given in Figure 3.1 contains non-standard syntax which is not found in C, as explained in Figure 1.3. Note that Booleans are also encoded as integers.

load and store statements are obtained while simplifying statements which interact with the heap, which is explained in Chapter 4. The statements malloc and calloc are also non-standard, as they can only allocate memory for a single value of the Type each time they are called. The Type must be passed using the syntax sizeof(Type), which is not shown in the grammar for simplicity.

assert and assume statements are part of the TriCera input language, and they are used to specify program properties.

Due to the way that structs are modeled in TriCera (as explained in Chapter 7, CIMP grammar does not allow writing to struct fields. The statements writing to struct fields are
instead reduced to statements which create a new `struct` value and assigning this value to the `struct` containing that field, as explained in Chapter 4. The compound literal expression given in the grammar is used for this purpose.

An example CIMP program which creates a doubly linked list, and adds some nodes to its tail, is given in Listing 3.1. `load` and `store` operations are used when interacting with the heap, and `struct` field updates are replaced with compound literal expressions. The compound literal expressions are used to transform `struct` field writes into direct updates to the variable holding the `struct` value, for reasons explained in Chapter 7.

```
struct node
{
  struct node *L;
  struct node *R;
};

void main()
{
  // allocate memory on the heap for list
  struct node* list = malloc(sizeof(struct node));

  // set the fields of list to 0, which are on the heap
  struct node tmp = load(list);
  tmp = (node){L = 0,     R = tmp.R};
  tmp = (node){L = tmp.L, R = 0    };  
  store(list, tmp);

  struct node *tail=list;

  int i = 0;
  while(i < 10) // will add 10 more nodes to the list
  {
    struct node *n = malloc(sizeof(struct node));

    int tmp2 = load(n);
    tmp2 = (node){L = tail,     R = tmp2.R};
    tmp2 = (node){L = tmp2.L, R = 0    };  
    store(n, tmp2);

    int tmp3 = load(tail);
    tmp3 = (node){L = tmp3.L, R = n};
    store(tail, tmp3);

    tail = n;
    i = i + 1;
  }
  assert(list != tail);
}
```

Listing 3.1: An example CIMP program
CHAPTER 3. SYNTAX OF CIMP
Chapter 4

Simplification of TriCera Parsable Programs into CIMP Programs

Figure 3.1 omits the syntax for assignment to struct fields, pointer accesses and some expressions or statements which can be considered syntactic sugar, which are actually parsed by TriCera. In this section, this simplification process from the TriCera parsable language into CIMP is explained.

Note that the actual starting point is a subset of the TriCera parsable syntax. Some common syntax such as for loops, function calls etc. are omitted, as the focus of this thesis is on the modeling of Heap.

The simplification process consists of three stages,

- desugaring (i.e. simplification of syntactic sugar syntax),
- simplification of assignments to struct fields,
- simplification of pointer accesses.

The three simplification stages are applied repeatedly until a fixed point is reached, i.e. no further simplification is possible. The semantic analysis starts after this simplification stage.

4.1 Desugaring

The expressions / statements which can be expressed using other expressions / statements in the grammar are given in Table 4.1.

4.2 Simplification of Assignments to struct Fields

A struct data type with n fields can be shown simply as the n-tuple: \( \langle f_i : Type_i \rangle_{i \in 1..n} \), where each field has a unique label expressed as \( f_i \) and has an associated type. A struct value is defined in the syntax as \( \langle f_i : v_i \rangle_{i \in 1..n} \).

In accordance with the syntax, the struct data type is represented using algebraic data types (ADTs) in TriCera. Consequently, this means that the fields are not directly addressable in memory, but accesses must go through the owner of the field, the parent struct. This also means that if the value of a field is updated, a new struct value must be created where the only change from the previous struct is the updated field, and this struct value must be used to update the value in memory where the original struct was
4.2. ASSIGNMENT TO FIELDS  

<table>
<thead>
<tr>
<th>Syntactic sugar syntax</th>
<th>CIMP syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_1 += e_2 )</td>
<td>( e_1 = e_2 + e_1 )</td>
</tr>
<tr>
<td>( e_1 -= e_2 )</td>
<td>( e_1 = e_2 - e_1 )</td>
</tr>
<tr>
<td>( e_1 *= e_2 )</td>
<td>( e_1 = e_2 * e_1 )</td>
</tr>
<tr>
<td>( e_1 /= e_2 )</td>
<td>( e_1 = e_2 / e_1 )</td>
</tr>
<tr>
<td>( e_1 %= e_2 )</td>
<td>( e_1 = e_2 % e_1 )</td>
</tr>
<tr>
<td>( x \rightarrow f )</td>
<td>((x).f)</td>
</tr>
<tr>
<td>( e++ ) or ( ++e )</td>
<td>( e = e + 1; )</td>
</tr>
<tr>
<td>( e-- ) or ( --e )</td>
<td>( e = e - 1; )</td>
</tr>
</tbody>
</table>

Table 4.1: Conversion of syntactic sugar syntax into CIMP syntax. Note that this thesis only considers side-effect-free expressions, which means that assignments and pre/post-increment/decrement operators cannot be used as expressions; in other words they can only be statements. This means that although post-increment and pre-increment operators (i.e. \( e++ \) and \( ++e \)) have different semantics, they are considered the same during desugaring, as they can only be used as stand-alone statements. The same is true for post-decrement and pre-decrement operators.

Located. So, the goal of this simplification stage is to replace assignments to struct fields with assignments to structs.

Here a syntax that resembles a compound literal is used to simplify writes to struct fields, where a new struct value is created where the only changed field from the original struct is the field to which the assignment was done. Note that the compound literal syntax is (currently) not directly available in TriCera to create a new struct, and only produced as a means to simplify field writes.

There are two main differences from the actual compound literals of C99. Assignments done with the C99 compound literal is actually syntactic sugar for creating a temporary initialized struct variable and then assigning it to the actual left-hand side [31]. In CIMP, as stated before, only a single field value is changed automatically during simplification; and assignments to fields are evaluated as a single instruction rather than two different statements.

Direct and indirect assignments to struct fields will be explained using the code shown in Listing 4.1.

Listing 4.1: The code shows a struct of type Str, which contains an int field f and a pointer-to-int field pf. The struct has a single instance on the stack, x, with a stack pointer, ps, pointing to it. A single instance of Str is also allocated on the heap using malloc, with a heap pointer, ph, pointing to it.
4.2.1 Direct Assignments

When doing a direct assignment to a `struct` field (i.e. when the parent `struct` is on the stack, e.g. Listing 4.1, line 7), the simplification steps are:

- Create a new `struct` value, where the only difference is the field that the assignment was done to,
- Update the variable with the new value.

Both steps are achieved by using the following simplification rule, where the premise represents the syntax before simplification, and the conclusion represents the simplified syntax in CIMP. The replacement rule is given in Equation 4.1.

\[
e.f = v \quad \Rightarrow \quad e = (\tau)\{e.f_i \in 1..j-1, v, e.f_j k \in j+1..n\}; \quad (4.1)
\]

\(\tau\) represents the type of \(e\), i.e. \(e \in \text{dom}(\tau)\), and \(f\) is the \(j\)th field of the `struct`. In case of direct assignment, \(e\) can only be a non-pointer variable.

As an example, for the direct assignment shown in Listing 4.1, the following replacement takes place:

\[
x.f = 42;
\]

\[
to \quad x = (\text{Str})\{42, x.pf\};
\]

The new field value 42 is assigned to field \(f\), and field \(pf\) is assigned its original value from \(x\).

4.2.2 Indirect Assignments

When doing an indirect assignment to a `struct` field (i.e. when the parent `struct` is reached through a pointer indirection), the expression \(e\) in Equation 4.1 is a dereferenced pointer variable.

For the example indirect assignments shown in Listing 4.1, the following replacements take place:

\[
(*ps).f = 43;
\]

\[
to \quad (*ps) = (\text{Str})\{43, (*ps).pf\};
\]

\[
and \quad (*ph).f = 44;
\]

\[
to \quad (*ph) = (\text{Str})\{44, (*ph).pf\};
\]

Note that this simplification results in code which must be further simplified as explained in Section 4.3. This is because dereferenced pointer variables cannot be in the left-hand side of an assignment according to the grammar of CIMP.

4.2.3 Nested structs

Assignments to nested `struct` fields are simplified in the same way as regular `structs`, but from right to left (or bottom-up). An example is given in Table 4.3.
4.3 Simplifying Pointer Accesses

The last stage of simplification is for pointer accesses into `load & store` operations. This simplification is done only for pointers that point to the heap. `load & store` operations are normally not parsable by TriCera; but they are part of the syntax of CIMP as shown in Figure 3.1.

Before going into why this is only done for heap pointers, some clarification must be made as to what exactly heap and stack pointers are.

Pointers to global/static/local variables are called *stack pointers* in this document. In TriCera, these variables are modeled precisely for verification, i.e. no abstraction. Pointers to a location on the heap are called *heap pointers*. In TriCera, heap pointers are modeled at a higher level of abstraction resulting in a less precise verification procedure (i.e. not complete). A comparison of stack vs heap pointers is given in Table 4.2.

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>Complete(^1)</th>
<th>Declare uninitialized(^2)</th>
<th>Reassign(^3)</th>
<th>Point to struct fields(^4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap pointer</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.2: Stack vs heap pointers

How to differentiate between stack and heap pointers?

During simplification, in many cases it would be hard for the analyzer to understand whether the pointer points to the stack or to the heap. Currently, this is easily done in TriCera by limiting how the stack pointers can be used and initialized.

- The first constraint is that the stack pointers are always initialized at declaration, with the initialization value being the address of another variable on the stack.
- The second constraint is that the stack pointers cannot be reassigned. This ensures that the initialization value is kept throughout the program execution, which also means that the pointers cannot switch between being heap pointers and stack pointers.

These constraints mean that the stack pointers can only point to a single location on the stack (the initialization value), and that value cannot change during execution. This makes it possible to statically replace all pointers to the stack with the actual variables, without the need for a points-to analysis. This can be shown with the simplification rule below:

\[
\frac{\star sp}{x} \text{ where sp points to x}
\]

Only accesses to the heap remain after this simplification, which are converted into `load & store` operations as explained in the following sections.

\(^1\)The representation is fully precise.
\(^2\)Pointer variable can be declared uninitialized (e.g. `int *x;`).
\(^3\)An assignment can be done to the pointer variable after initialization.
\(^4\)Pointer variable can point to `struct` fields (e.g. `int *y = &x.f`).
CHAPTER 4. SIMPLIFICATION TO CIMP  
4.4. SIMPLIFICATION EXAMPLES

Reading From The Heap

For heap pointers, the pointed value is first loaded to a fresh variable (shown as temp in the rule below), then this variable is used to read the pointed value. The read operation is done using a load operation. The rule for this simplification is

\[
\begin{align*}
  x &= \ast hp \\
  \text{temp} &= \text{load}(hp); \quad x = \text{temp}.
\end{align*}
\]

A fresh variable is necessary because the load operation is a statement in CIMP as shown in Figure 3.1. This is done as the reduced code is simpler and semantically closer to C. Consider the example statement to be simplified

\[x = \text{node}\rightarrow\text{next}\rightarrow\text{data};\]

Above statement would reduce to

\[
\begin{align*}
  \text{tmp1} &= \text{load}(\text{node}); \\
  \text{tmp2} &= \text{load}(\text{tmp1}.\text{next}); \\
  x &= \text{tmp2}.\text{data};
\end{align*}
\]

Writing To The Heap

For heap pointers, the assignment is simply replaced with a store statement using the rule

\[
\begin{align*}
*hp &= x \\
\text{store}(hp,x).
\end{align*}
\]

4.4 Simplification Examples

Some examples for the simplifications described in this section are given in Table 4.3. The code for the examples is given in Listing 4.1. The simplifications are done until a fixed point is reached, meaning that no further simplification is possible. The column “Simplification steps” shows all steps including the intermediate ones; and only the green colored statements will appear in the output of the simplification stage.

Listing 4.2: A nested struct of type Nested

```
typedef struct FStr { int f1, f2; }
typedef struct Nested { FStr fs;
    int f; } ns;
```
4.4. SIMPLIFICATION EXAMPLES

CHAPTER 4. SIMPLIFICATION TO CIMP

Original statement

Assignments to struct fields

<table>
<thead>
<tr>
<th>Pointer accesses</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1st pass</th>
<th>2nd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps-&gt;f = 42</td>
<td>(*ps).f = 42</td>
</tr>
<tr>
<td>temp = load(ph); y = temp.f</td>
<td></td>
</tr>
<tr>
<td>x = (Str) {42, x.pf}</td>
<td></td>
</tr>
<tr>
<td>ph-&gt;f = 42</td>
<td>(*ph).f = 42</td>
</tr>
<tr>
<td>temp = load(ph); *(temp.pf) = 42</td>
<td></td>
</tr>
<tr>
<td>ns.fs.f1 = 42</td>
<td>(ns.fs) = (FStr) {42, ns.fs.f2}</td>
</tr>
</tbody>
</table>

Table 4.3: Code replacement examples which involve reading from & writing to struct fields. The initialization code used for the examples is given in Listing 4.1. Variable declarations are not shown for brevity (e.g. int y, STR tmp etc.). The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter. The simplifications are done from left to right, and if a simplification stage is applied several times those are indicated as 1st pass, 2nd pass etc. The columns show the simplification steps explained in this chapter.
Chapter 5

Semantics of CIMP

The semantics of CIMP statements are given in this chapter using structural operational semantics [32]. Note that the given semantics are not for the full C language, as the grammar of CIMP allows only a limited subset of C.

Since the expressions in CIMP do not have side effects, their evaluation is done with the eval function given in Section 5.2.

The inference rules are given as transitions from one configuration to another, i.e. $C \rightarrow C'$. Configurations consist of a statement to be executed, and the program state which consists of the stack $s^1$ and the heap $h^2$. $s$ can be defined as the function mapping variables to values, $s = x \mapsto v$. In all the rules, $x$ and $x_i$ refer to variables located on the stack $s$

$$s = (x_1 \mapsto v_1, x_2 \mapsto v_2, ..., x_n \mapsto v_n).$$

The heap is modeled using locations (l), which represent memory addresses on the heap. Locations are implemented as integer values. Since TriCera currently does not support pointer arithmetic, each data type is modeled as occupying only a single location on the heap. $h$ can be defined as the partial function mapping locations to values, $h = l \mapsto v$.

The partial function is undefined for a location which is not allocated yet (using malloc or calloc).

$$h = (l_1 \mapsto v_1, l_2 \mapsto v_2, ..., l_n \mapsto v_n).$$

Every value $v$ in $s$ and $h$ also has a corresponding type, i.e. $v_i \in \text{dom}(\text{Type}_i)_{i=1..n}$.

For $\mu \in \{s, h\}$, the notation $\mu[\alpha \mapsto \beta]$ means that, $\alpha$ is mapped to $\beta$, and all other locations in $\mu$ are unchanged. The notation $\mu(\alpha)$ is used to get the value mapped to $\alpha$ in $\mu$.

5.1 Statements

The inference rules for the statements are applied until the final configuration $C_f = (\text{skip}, (s, h))$, or the special error configuration Error is reached. The following sequential composition rules are applied to reduce statements:

$$
(S_1, (s, h)) \rightarrow (S'_1, (s', h')) \quad (S_1; S_2, (s, h)) \rightarrow (S'_1; S_2, (s', h')) \quad (\text{skip}; S, (s, h)) \rightarrow (S, (s, h))
$$

Some rules require the evaluation of an expression (i.e. $eval_s(e)$). The function for evalu-
5.1. STATEMENTS

5.1.1 Assignment

Assignment and allocation is only possible when there is a variable as the left-hand side as stated in the grammar of CIMP; so there is a single rule for assignment, which is given in Table 5.1.

\[
\text{[assign]}: \text{assignment to variable } (x = e, (s, h)) \rightarrow (\text{skip}, (s[x \mapsto \text{eval}_s(e)], h))
\]

Table 5.1: Rule for regular assignment

5.1.2 Branching Statement (if-else) & Looping Statement (while)

Table 5.2 shows how the branching and the looping statement are handled.

For the branching statement, one of the two branches are executed depending on the evaluation value of the guard ([if.1] or [if.2]). The looping statement is simply converted into a branching statement which has a compound statement in the true branch that contains the initial statement, and a skip statement in the false branch [while]. Then one of the rules [if.1] or [if.2] are applied, and possibly loop if [if.1] is taken. Since it is recursive, the program might run forever if \( e \) always evaluates to true.

\[
\begin{align*}
\text{[if.1]} &: \text{true branch } (\text{if}(e) S_1 \text{ else } S_2, (s, h)) \rightarrow (S_1, (s, h)) \quad \text{eval}_s(e) \neq 0 \\
\text{[if.2]} &: \text{false branch } (\text{if}(e) S_1 \text{ else } S_2, (s, h)) \rightarrow (S_2, (s, h)) \quad \text{eval}_s(e) = 0 \\
\text{[while]} &: (\text{while}(e) S, (s, h)) \rightarrow (\text{if}(e) (S; \text{while}(e) S) \text{ else } \text{skip}, (s, h))
\end{align*}
\]

Table 5.2: Branching (if-else) and looping (while) statements

5.1.3 Heap Related Statements (load, store, malloc & calloc)

Rule [load] is used to read values from heap locations, rule [store] is used to assign values to heap locations, and rules [malloc] & [calloc] are used to allocate uninitialized & initialized memory respectively. [load] uses the evaluation function \( \text{eval} \), which is defined at Section 5.2.

\( \text{zero} (\tau) \) which is used in Table 5.3 maps a value to the type \( \tau \). It returns a 0 value if the type \( \tau \) is an integer, or a struct of type \( \tau \) with all its fields recursively initialized to 0 if \( \tau \) is a struct. It can be defined more formally as the function given in Equation 5.1.

\[
\text{zero} (\tau) = \begin{cases} 
0 & \tau \text{ is of type int} \\
\{ f_i \mapsto \text{zero}(\tau_i) \} \quad & \tau \text{ is of type } \langle f_i : \tau_i \rangle_{i \in \{1..n\}} \text{ (i.e. a struct type)} 
\end{cases}
\]

5.1.4 assert & assume Statements

Table 5.4 shows how the functions assert & assume are handled. [assert.1] states that the assert function has no effect on the program execution if the predicate evaluates to
CHAPTER 5. SEMANTICS

5.1. STATEMENTS

[load] \[\text{eval}_s(e) \in \text{dom}(h) \implies (x = \text{load}(e), (s, h)) \rightarrow (\text{skip}, (s[x \mapsto \text{eval}_s(e)], h))\]

[store]: store operation \[\text{store}(e_1, e_2), (s, h) \rightarrow (\text{skip}, (s, h[\text{eval}_s(e_1) \mapsto \text{eval}_s(e_2)]))\]

[malloc]: uninitialized allocation \[x = \text{malloc}(	au), (s, h) \rightarrow (\text{skip}, (s[x \mapsto l], h \uplus [l \mapsto v]))\]

[calloc]: zero initialized allocation \[x = \text{calloc}(	au), (s, h) \rightarrow (\text{skip}, (s[x \mapsto l], h \uplus [l \mapsto \text{zero}(	au)]))\]

Table 5.3: load, store, malloc and calloc

true (i.e. \( n \neq 0 \)), and the error configuration \textbf{Error} is reached if the predicate evaluates
to false (i.e. \( n = 0 \)).

[assume] rule converts \textbf{assume} statements into \textbf{while} statements with an empty body
and a negated predicate. This is equivalent to saying that the program does not proceed
until the predicate of the \textbf{assume} function holds.

[assert.1] \[\text{assert}(e), (s, h) \rightarrow (\text{skip}, (s, h)) \implies \text{eval}_s(e) \neq 0\]

[assert.2] \[\text{assert}(e), (s, h) \rightarrow \textbf{Error} \implies \text{eval}_s(e) = 0\]

[assume] \[\text{assume}(e), (s, h) \rightarrow (\textbf{while}(\neg e) \text{ skip}, (s, h))\]

Table 5.4: assert \& assume statements
5.2 Expressions

The expressions in TriCera evaluate into values ($v$). Since none of the expressions in CIMP contain any side effects, a recursive evaluation function $eval_s$ can be defined for this purpose. $eval_s$ is given in Equation 5.2. It takes the expression to be evaluated as its argument, and the evaluation is done using the stack $s$. $eval_s$ calls two other functions, namely $evalBinOp$ and $evalUnOp$ to evaluate expressions containing binary and unary operations respectively. $evalBinOp$ is given in Equation 5.3 and $evalUnOp$ is given in Equation 5.4.

The ternary operator notation from C is used to define the result in some cases of $evalBinOp$ and $evalUnOp$. This was done in order to return the values as integers, as CIMP does not support Booleans. If the predicate to the left of "?" holds, then the integer "1" is returned (left-hand side of the colon), otherwise the integer "0" is returned (right-hand side of the colon).

\[
 eval_s(e) = \begin{cases} 
 s(x) & \text{for } e = x \\
 eval_s(eval_s(e_1).f) & \text{for } e = e_1.f \text{ and } e_1 \neq \langle f_i \mapsto v_i \; i \in 1..n \rangle \\
 v_j & \text{for } e = \langle f_i \mapsto v_i \; i \in 1..n \rangle.f_j \\
 evalBinOp(op,eval_s(e_1),eval_s(e_2)) & \text{for } e = e_1 \text{ op } e_2 \\
 evalUnOp(op,eval_s(e_1)) & \text{for } e = \text{ op } e_1
\end{cases}
\]

(5.2)

\[
 evalBinOp(op,n_1,n_2) = \begin{cases} 
 n_1 + n_2 & \text{for } op = "\ +" \\
 n_1 - n_2 & \text{for } op = "\ -" \\
 n_1 \cdot n_2 & \text{for } op = "\ \ast" \\
 n_1/n_2 & \text{for } op = "\ /" \\
 n_1 \text{ mod } n_2 & \text{for } op = "\ %" \\
 n_1 \geq n_2 ? 1 : 0 & \text{for } op = "\geq" \\
 n_1 < n_2 ? 1 : 0 & \text{for } op = "\ <" \\
 n_1 \leq n_2 ? 1 : 0 & \text{for } op = "\leq" \\
 n_1 = n_2 ? 1 : 0 & \text{for } op = "\==" \\
 n_1 \neq n_2 ? 1 : 0 & \text{for } op = "\!=" \\
 (n_1 \neq 0) \land (n_2 \neq 0) ? 1 : 0 & \text{for } op = "\&\&" \\
 (n_1 \neq 0) \lor (n_2 \neq 0) ? 1 : 0 & \text{for } op = "\|\|
\end{cases}
\]

(5.3)

\[
 evalUnOp(op,n) = \begin{cases} 
 -n & \text{for } op = "\ -" \\
 n = 0 ? 1 : 0 & \text{for } op = "\ !"
\end{cases}
\]

(5.4)
Chapter 6

Horn Clauses for Basic Programs

This chapter intends to explain the encoding of basic CIMP programs into Horn clauses, by introducing the toHorn function. Basic CIMP programs are those which do not contain structs nor heap interactions. Translation of programs containing structs will be explained in Chapter 7, and translation of programs containing heap interactions in Chapter 8.

6.1 Overview

There are various ways of translating programs into Horn clauses. A straightforward way is to define the formal operational semantics of the language, and then translate these semantics into a constraint logic program (i.e. CHCs) [33]. This method seems to be popular in the constraint logic programming community [8], [34].

It is also possible to directly encode the programs in Horn clauses without defining the semantics [35]. [13] shows how Horn clauses can be directly generated from the control flow graph (or the program graph) of programs.

The approach taken in this report is similar to the first approach, where the clauses are automatically generated from the formal operational semantics of the CIMP as defined in Chapter 5. For same translations such as the looping statement, a direct translation from CIMP programs into Horn clauses is also used as it is much more straightforward.

Note that translation of functions into Horn clauses is not considered in this chapter, as translation of functions is outside the scope of this work.

6.2 Translation into Horn Clauses

The translation into Horn clauses is formalized with the recursive toHorn function which contains cases for every possible statement in CIMP. The toHorn function for basic programs is given in Figure 6.1.

To translate the rules into Horn clauses, each statement is assigned an entry predicate $P_{entry}$ and an exit predicate $P_{exit}$. The entry predicate reflects the configuration before the statement executes, and the exit predicates reflects the configuration after.

$$(S; s) \rightarrow^{*} (\text{skip}; s')$$

which translates into

$$P_{entry}(s) \rightarrow P_{exit}(s')$$

“$(S; s) \rightarrow^{*} (\text{skip}; s')$” says that a statement starting in state $s$ will eventually (in zero or more steps of execution) reach the final configuration, which will contain the new state $s'$. In Horn clauses this relates to the two predicates $P_{entry}$ and $P_{exit}$, with the states being the
6.2. TRANSLATION

CHAPTER 6. HORN CLAUSES - BASIC

**Figure 6.1**: \( \text{toHorn} \) function for basic programs. Note that this is not the actual Horn clause notation in TriCera. TriCera outputs the clauses in Prolog notation, and the stack variables are unrolled whereas here the stack is simply shown as \( s \).

\[
\text{toHorn}(S, P_{\text{entry}}, P_{\text{exit}}) =
\begin{align*}
(1) \quad & \text{toHorn}(S_1, P_{\text{entry}}, P_{\text{new}}) \\
& \cup \text{toHorn}(S_2, P_{\text{new}}, P_{\text{exit}}) \\
& S_1; S_2
\end{align*}
\]

\[
(2) \quad \{ P_{\text{entry}}(s) \rightarrow P_{\text{exit}}(s[x \mapsto \text{exp}(e)]) \} \\
\{ P_{\text{entry}}(s) \wedge \text{exp}(e) \neq 0 \rightarrow P_{\text{entry}}S_1(s) \} \\
\cup \{ P_{\text{entry}}(s) \wedge \text{exp}(e) = 0 \rightarrow P_{\text{entry}}S_2(s) \} \\
\cup \text{toHorn}(S_1, P_{\text{entry}}S_1, P_{\text{exit}}) \\
\cup \text{toHorn}(S_2, P_{\text{entry}}S_2, P_{\text{exit}}) \\
S_1; S_2
\]

\[
(3) \quad \{ P_{\text{entry}}(s) \wedge \text{exp}(e) \neq 0 \rightarrow P_{\text{exit}}(s) \} \\
\{ P_{\text{entry}}(s) \wedge \text{exp}(e) = 0 \rightarrow P_{\text{exit}}(s) \} \\
\text{if}(e) \ S_1 \ \text{else} \ S_2
\]

\[
(4) \quad \{ P_{\text{entry}}(s) \wedge \text{exp}(e) \neq 0 \rightarrow P_{\text{Body}}(s) \} \\
\cup \{ P_{\text{entry}}(s) \wedge \text{exp}(e) = 0 \rightarrow P_{\text{exit}}(s) \} \\
\cup \text{toHorn}(S_{\text{Body}}, P_{\text{Body}}, P_{\text{entry}}) \\
\text{while}(e) \ S_{\text{Body}}
\]

\[
(5) \quad \{ P_{\text{entry}}(s) \wedge \text{exp}(e) = 0 \rightarrow \text{false} \} \\
\cup \{ P_{\text{entry}}(s) \wedge \text{exp}(e) \neq 0 \rightarrow P_{\text{exit}}(s) \} \\
\text{assert}(e)
\]

\[
(6) \quad \{ P_{\text{entry}}(s) \wedge \text{exp}(e) \neq 0 \rightarrow P_{\text{exit}}(s) \} \\
\text{assume}(e)
\]

\[
(7) \quad \{ P_{\text{entry}}(s) \rightarrow P_{\text{exit}}(s) \} \\
\text{skip}
\]

The translation produces one or more Horn clauses as its output. The “\( \cup \)” operator is used to extend the set of created Horn clauses.

The function \( \text{exp} \) used in some of the translation cases is similar to the \( \text{eval} \) function given in Equation 5.2; however, the expressions are not really reduced to a value by \( \text{exp} \), which is the case with \( \text{eval} \). This is because expressions without side effects can be almost directly translated into literals and constraints that are part of Horn clauses. \( \text{exp} \) function is given in Figure 6.2.
\[
\begin{align*}
\text{exp}(e) = \begin{cases}
  x & \text{if } x \\
  n & \text{if } n \\
  l & \text{if } l \\
  f(\text{exp}(e)) & \text{if } e.f \\
  \text{Type}(\text{exp}(e_1), ..., \text{exp}(e_n)) & \text{if } (\text{Type})\{f_i \mapsto e_i \mid i \in 1..n\} \\
  (\text{exp}(e_1) \text{ BinOp } \text{exp}(e_2)) & \text{if } e_1 \text{ BinOp } e_2 \\
  \text{UnOp } \text{exp}(e) & \text{if } \text{UnOp } e
\end{cases}
\end{align*}
\]

Figure 6.2: \text{exp} function translates side effect free expressions of CIMP into logic literals or constraints which are part of the generated Horn clauses by \text{toHorn}. Note that there is no translation for \text{struct} values, as \text{struct}s are represented by ADTs in Horn clauses and variables are not reduced to values. The only expression which generates \text{struct} values is the expression \((\text{Type})\{f_i \mapsto e_i \mid i \in 1..n\}\), which is translated into an ADT constructor. For details on ADTs see Chapter 7. \text{BinOp} and \text{UnOp} used in the last two cases are given in Figure 3.1; however, note that the actual syntax of these operators might be different in Horn clauses after translation.

Each translation case is explained in the following paragraphs.

**Sequential Composition (1)** If the translated statement has other statements coming after it, the exit predicate can be merged into the next statement’s entry predicate. This also means that if the statement has other statements coming before it, the entry predicate can be merged into the previous statement’s exit predicate. If the statement evaluates into more statements, then these are enclosed within the entry and the exit predicates. (1) in \text{toHorn} achieves this by recursively translating statements into clauses and creating new entry / exit predicates which does the merging.

**Assignment Statement (2)** The semantics of the assignment statement was given in Table 5.1. (2) is a direct translation of this semantic rule, which simply updates the stack in the exit predicate.

**Branching Statement (3)** The semantic rules for the branching statement are given in Table 5.2.

Since each branch contains another statement (\(S_1\) and \(S_2\)), they must be translated recursively, and a fresh predicate is created for each branch (\(P_{ entryS_1}\) and \(P_{ entryS_2}\)).

Branching rules also contain constraints, which can be expressed in conjunction with the bodies of the Horn clauses:

\[
P_{ entry}(s) \land \text{exp}(e) \neq 0 \implies P_{ entryS_1}(s) \quad P_{ entry}(s) \land \text{exp}(e) = 0 \implies P_{ entryS_2}(s)
\]

**Looping Statement (4)** The looping statement is converted to a branching statement as defined in [while]; however, here it will be directly defined in Horn clauses as it is more straightforward.

The case states that if the predicate evaluates to something other than zero, the body statement (\(S_B\)) is executed, otherwise the loop ends. The body statement’s entry predicate is defined as (\(P_B\)). Since the loop is repeated after the body is executed, the exit predicate for \(S_B\) is \(P_{ entry}\).

**assert Statement (5)** As shown in the [assert] rule given in Table 5.4, the \text{assert} statement can be considered as a branching statement. The \text{true} branch does not effect
the output (i.e. same as skip, and the false branch leads into the Error configuration. The Error configuration in Horn clauses is defined as false, so the false branch should transition into false. This leads to the two clauses generated by (5).

**assume Statement (6)** The assume statement means the transition to the next configuration can only happen if the predicate evaluates to true, and no progress is done if the predicate evaluates to false. In Table 5.4 this was done with converting the assume statement into a while statement which loops (with a skip statement) until the predicate evaluates to true. In Horn clauses, the progress can simply be prevented by adding a constraint to the clause body as shown in (6).

**skip Statement (7)** The skip statement simply transitions to the next configuration without changing the state, which in turn translates to the Horn clause shown in (7).
Chapter 7

Horn Encoding of C structs Using Algebraic Data Types

Chapter 6 discussed the encoding of basic CIMP programs into Horn clauses, by introducing the toHorn function. This chapter adds the translation cases for the C struct type.

C struct is a composite data type. It is implemented as contiguous blocks of physical memory, whose size depends on the number of fields the struct has. The C struct is mutable, that is, the values of its fields can be modified after creation.

structs in TriCera are modeled using the theory of algebraic data types (ADTs), which is provided by ELDARICA [6]. However, ADT objects are immutable, which means that a new object must be created each time a struct field is modified, and then the whole struct value must be updated using this object.

This chapter intends to first give more information about ADTs, and then explain how the C struct can be modeled using them.

7.1 Algebraic Data Types (ADTs)

Algebraic data types (not to be confused with abstract data types) make it possible to easily specify custom data types. They are available in many higher level languages such as Haskell, OCaml and Scala. ADTs are constructed using a combination of sums and products.

Sums are variant types, where the instance can only hold a single value which is a variant of the defined types. E.g. consider the simple example

```c
enum week {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The example uses the enum data type in C, which is a sum type. The instances of the type week can only hold one value from the given list of values. The sum part comes from the fact that the number of different values possible for this data type is the sum of its elements, which in this case would be 7.

Products are composite types such as tuples, records, or the struct type in C. A constructor is used to combine the values contained by the product type into a single value, and a selector is used to select one of the contained values. Similarly, the product part comes from the fact that the number of different values possible for this composite data type is the products of the values possible for its elements. Products are usually shown using a Cartesian product of the contained values.
7.2 The C struct

The C struct is a product type of the form \((f_1 : \tau_1 \times f_2 : \tau_2 \times \ldots \times f_n : \tau_n)\), where \(f_n\) and \(\tau_n\) represent the field label and the field type respectively.

When one of the fields of the struct is read (not updated), this is equivalent to getting the selector from the ADT for that field, and applying it to its parent struct.

Consider the example code given in Listing 7.1. The access to \(f_1\) at line 3 is thus turned into the function application \(f_1(x)\), since \(x\) is the parent of \(f_1\).

When a struct field is assigned a value, this means that the struct containing that field (i.e. its parent) must be updated. In this case a new term of the field’s parent type (and parent’s parents if it is nested) is created, with the updated field replaced with the assigned value. This operation is similar to the simplification done in Chapter 4.2; where the new ADT term in Horn clauses corresponds to the new struct value created using the compound literal syntax.

In the example code given in Listing 7.1, the assignment in line 2 would result in the new term \(S(42,f_2(x))\) to be assigned as the new value of \(x\).

```c
struct S { int f1, f2; } x;
int y;
x.f1 = 42; // line 1
y = x.f1;  // line 2
```

Listing 7.1: Accessing the fields of a simple struct.

In Horn clauses, line 1 would be encoded as

\[
P_{entry}(x,y) \rightarrow P_{exit}(S(42,f_2(x)),y)
\]

and line 2 would be encoded as

\[
P_{entry}(x,y) \rightarrow P_{exit}(x,f_1(x)).
\]

7.3 Nested structs

Nested structs are modeled in the same way as unnested structs. However, a translation case for nested structs will not be defined, as they are actually reduced to simple struct assignments during simplification (see Table 4.3 for an example).

When reading a field from a nested struct field, the outermost term is the field being accessed, and the innermost term is the root struct as was the case with unnested structs.

When an assignment is done to a nested field, the final term is constructed in a recursive way. To illustrate this, consider the code given in Listing 7.2 and the struct shown as a tree in Figure 7.1. To construct the final term, the following steps are taken:

- ns calls its child s2 to build a new term for itself. It also passes its own term to its child.
- s2 creates the new term \(S(f_1(s_2(ns)), 42)\) and returns it to its parent. This is a new term of the type \(S\) (s2’s type). Note that \(f_2\) is updated with the new value, and all other field values are preserved as they are loaded from the unmodified struct ns (in this case \(f_1(s_2(ns))\)), which is possible as s2 was passed its parent term.
- ns gets the new term from its child s2, and builds the final term \(Nested(s_1(ns), S(f_1(s_2(ns)), 42))\).
This is similar to the previous step. \( s2 \)'s sibling \( s1 \) is unmodified, so it is used directly, and \texttt{Nested} is the type of the root term.

\begin{verbatim}
struct S { int f1, f2; };
struct Nested { struct S s1, s2; } ns;
ns.s2.f2 = 42;
\end{verbatim}

Listing 7.2: Updating a nested \texttt{struct} field.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig7_1}
\caption{A nested struct as a tree, with the assignment \texttt{ns.s2.f2 = 42}.}
\end{figure}
Chapter 8

Horn Encoding of Pointers

Chapter 6 discussed the encoding of basic CIMP programs into Horn clauses, by introducing the $toHorn$ function, and Chapter 7 expanded on it by explaining the encoding of C structs. This chapter adds the final translation cases for the $toHorn$ function, which are used to translate statements that interact with the heap.

8.1 Horn Encoding of Heap Pointers

8.1.1 Overview

The semantic rules for heap related statements [load], [store], [malloc] and [calloc] are given in Table 5.3. In these rules the heap is modeled as another store $h$, similar to how the stack $s$ is modeled; however, it is challenging from a verification point of view to model the heap this way (see Chapter 2.2). Instead, TriCera uses a similar abstraction methodology as described in JayHorn [16].

In JayHorn, the concept of space invariants are introduced, which are automatically computed during verification. Heap interactions are abstracted using these space invariants [24]. The main idea is that instead of modeling each heap location precisely, the invariant models the properties that hold for the heap at each program location.

To give a very simple example, consider the following program given in Listing 8.1.

```c
1  x = malloc (sizeof(int));  // l1
2  assert(*x == 0);           // l2
3  *x = 42;                   // l3
4  y = *x;                    // l4
5  assert(y == 0 || y == 42); // l5
```

Listing 8.1: A very simple program with heap allocation and access

Considering the program starts at l1 (i.e. no prior heap interactions), the memory allocation at l1 means that all the values at the heap must be 0, thus the assertion at l2 should succeed. However, without flow-sensitivity, the assignment in l3 affects this assertion and causes it to fail. The assignment at l3 changes the heap invariant, meaning that the value $x$ points to can be 42 in addition to the value 0.

The assertion at l5 holds for the reason explained in the previous paragraph. The imprecisionness of the abstraction here is that the assertion $\text{assert}(y == 42)$ would fail at l5, which should actually hold as the old value (0) of $\text{*x}$ cannot reach this location. Some refinements are possible to increase the precision which would make this assertion succeed as well, and these will be discussed in Section 8.1.3.
8.1. HEAP POINTERS

Figure 8.1: push and pull operations. The top part shows that pushes are used to store the changes, and that pulls are used to load from the heap. The left part shows that each type is currently represented by a single invariant, so in this case all integer variables are mapped to inv_int. The part labeled Push shows step by step what happens when a new value is to be written to a location on the heap, and the part labeled Pull shows the steps when a value is to be read from the heap. Note that the variable tmp is actually assigned an arbitrary value at each load operation, before assuming the invariant, which is not shown in the diagram. If for example the pull operation is inside a loop, this ensures that the variable tmp holds a new fresh value at each iteration. This function which assigns an arbitrary value is usually called the havoc function.

This is a very abstract way of modeling the heap; especially without the refinements. However, even in its current stage, this abstraction is good enough for many C programs with shallow properties, as evidenced by the results of the experiments given in Chapter 10. Compared to Java programs, less precise modeling of the heap is also more acceptable for C programs, as C programs usually rely on stack pointers rather than heap pointers [26].

8.1.2 Method

In JayHorn, heap accesses are converted into pull and push operations [24]. The idea is similar to working with the heap as a repository, where the latest version is read using a pull, and the changes are submitted using a push. In JayHorn, the pull operation reads all the fields of an object from the heap by assuming the heap invariant; and a push operation writes back all the fields of an object to the heap by asserting the heap invariant.

In TriCera, the behavior of pulls and pushes are similar to the behavior in JayHorn; however, the pulls and pushes are done using a single variable instead of a number of fields, since C does not contain objects as in Java. structs are also modeled using ADTs as explained in Chapter 7, so the whole ADT value is pushed/pulled instead of the individual fields. A simple depiction of the push and pull operations is given in Figure 8.1.

In the encoding, a heap invariant is created for each distinct data type that was allocated memory on the heap. E.g. a single invariant for Int is created, and all structs of the same type share a single invariant as well. This is not very precise; however, methods will be described to increase precision in Section 8.1.3.

Some rules must first be defined to translate C programs into programs containing push/pull operations. The simplifications rules given in Chapter 4 actually help in this regard, by first reducing the heap pointer accesses into load & store operations. E.g. the program given in Listing 8.1 is first reduced to the CIMP program given in Listing 8.2.
CHAPTER 8. HORN CLAUSES - POINTERS

8.1. HEAP POINTERS

![Heap representation as a partial function which maps locations to values](a)

![Heap invariants which are created for each type](b)

Figure 8.2: Representing the heap using heap invariants

```cimp
1 x = calloc(sizeof(int));  // 11
2 temp = load(x);
3 assert(temp == 0);  // 12
4 store(x, 42);  // 13
5 temp1 = load(x);
6 temp2 = load(x);
7 assert(temp1 == 0 || temp2 == 42);  // 14
```

Listing 8.2: The program from Listing 8.1 simplified into a CIMP program

The CIMP program in Listing 8.2 only contains load and store operations as heap accesses, which are the pull and push operations of TriCera.

Heap Invariants

As discussed earlier, a heap invariant $\phi$ is created for each data type on the heap, which captures the properties for that data type. For CIMP, this means the possible heap invariants are $\phi_{\text{int}}$ and one invariant for each struct type. Invariants for pointer types are also possible. The representation of heap as a partial function which maps locations to values, and the corresponding heap invariants are given in Figure 8.2.

Heap invariants in TriCera are predicates with an arity of two, and they take as arguments a heap location and the value stored at that location. The invariant is automatically inferred with the help of the pushes (assertions) and pulls (assumptions) done to / from the invariants.

To keep track of heap locations, a new global stack variable $H$ is created automatically when the program contains heap interactions (global means that it is added to every generated Horn clause). Whenever a memory allocation function is called, the location value is taken from this variable and the variable is incremented to differentiate between different memory locations. Since it is not possible to free allocated memory in TriCera, this location counter never decreases.
8.1. HEAP POINTERS

8.1.2 calloc & malloc

The memory allocation functions calloc and malloc both take a location value from the heap location counter \( H \) and then increment it. The location value is assigned to the pointer variable at the left-hand side.

**calloc**  zero initializes the newly allocated memory, which means an assertion must also be done to add the property to the invariant for the allocated type. Since Type can also be a non-trivial type such as an ADT (i.e. struct type), a getZero\( (Type) \) function is defined in Equation 8.1 which is analogous to the zero function defined in Equation 5.1 for the semantic rule [calloc]. It takes as its input a type name \( \tau \), and returns a zero initialized value. In case of a struct type, an ADT is recursively created with the fields initialized to zero. The translation case for calloc is given in (10) in toHorn as part of Equation 8.3.

**malloc** does not initialize the newly allocated memory to any value, thus the allocated memory can contain any value possible for that type. This of course causes all precision to be lost regarding value properties for that heap invariant, since an assertion must still be done to push the property. With refinements, it is possible to increase precision even for allocations using malloc; however, it is recommended that calloc is used instead.

To assert the property which states that the value can be anything, a fresh variable \( (x_{\text{fresh}}) \) is created and used in the assertion. The translation case for malloc is given in (11) in toHorn as part of Equation 8.3.

**store** simply asserts a new property for the invariant, using the passed value, as was done with the memory allocation functions. The translation case for store is given in (9) in toHorn as part of Equation 8.3.

**load** The load operation pulls a value from the heap, and this value must satisfy the previously asserted properties for the invariant. To achieve this, a fresh variable is created and assigned to the left-hand side of the load operation. Then to make this variable satisfy the previously asserted properties, the relevant heap invariant is assumed with the new fresh variable as a value for that location. This ensures that progress is only possible if the newly created variable satisfies the properties of the invariant.

The translation case for load is given in (8) in toHorn as part of Equation 8.3.

\[
\text{getZero}(\tau) =\begin{cases} 
0 & \tau \in \text{dom}(\text{int}) \\
\tau(\text{getZero}(\tau_1),...,\text{getZero}(\tau_n)) & \tau \in \text{dom}((f_i : \tau_i \ i \in 1..n)) 
\end{cases} \quad (8.1)
\]

8.1.3 Refinements

There are several refinements possible to increase the precision of verification [24], which will be explained in this section. All the optimizations require doing data-flow analysis, either on CIMP or the generated Horn clauses.

**Optimizing Placement of Push and Pull Operations**

This is the first optimization which was done with JayHorn, and it is the simplest possible optimization possible in TriCera as well. The idea is to reduce the number of push and pull operations (or load and store operations) by merging pulls done from the same location, and merging pushes done to the same location. When there are pulls and pushes for different pointer variables, it is usually required to first understand if these pointer variables are pointing to the same location (alias), before their placements can be optimized. This
requires doing a points-to analysis to determine which pointers point to the same location, and the precision of the analysis mostly determines the success of this optimization.

It has the same idea as working with a version control repository such as git, where it does not make sense to pull from or push to the main repository after each local change. Since the heap is modeled in an abstract way, as soon as a push is done, the precision of the pulled variable is lost, as it is cleaned from the stack.

Consider the program given in Listing 8.1. Although the value pointed by x was updated to be 42 at line 3, the assertion `assert(*x == 42)` would fail without this optimization, since the simplified code given in Listing 8.2 stores the value of x and pulls it again.

An optimized version of the code using this method is given in Listing 8.3. It removes loads followed by a store, and moves the store as far down as possible, making it possible for the assertion `assert(*x == 42)` to succeed.

```c
x = calloc(sizeof(int))
temp = load(x);
assert(temp == 0);
temp = 42;
assert(temp == 0 || temp == 42);
store(x, temp);
```

Listing 8.3: The program from Listing 8.1 simplified into a CIMP program, with optimal placement of push and pull operations

Adding Flow Sensitivity

Another optimization is to add flow sensitivity to the heap invariants [24]. The goal is to determine which pulls can be affected by which pushes, and increase precision by using only the properties from those push sites. This optimization would for example be able to eliminate the imprecision coming from `malloc`, if it is determined to be not the latest push for a pull site. Again data-flow analysis is required to generate the set of affecting pushes for each pull, and precision of the analysis determines the success of this optimization.

8.2 Horn Encoding of Stack Pointers

Stack pointers, as they are currently quite constrained in TriCera, do not require a special encoding. This is because they are reduced to the stack variables that they are pointing to during the simplification stage explained at Chapter 4.
8.2. STACK POINTERS

Figure 8.3: The \texttt{toHorn} function for programs which contain heap interactions. Note that it is assumed the function has access to the stack $s$. A heap location counter $H$ is added as a variable on the stack (shown as $H$ instead of $s$) for simplicity. Each location $l$ is also associated with a type (which is held in the pointer holding the location), and this is where the invariants are obtained from. The \texttt{havoc} function assigns an arbitrary integer value to its argument. Note that \texttt{havoc} is actually not in the grammar of CIMP; however, it is used here to express the fact that the return value after the call contains arbitrary values.

\begin{align*}
(\texttt{malloc}) & = x & (\texttt{malloc}L) & = \{1 + H = H ; H = x ; ((\texttt{havoc}f \, x \, H)^{\texttt{assert} \, \texttt{assume} \, \texttt{havoc}}) \, \texttt{malloc} \} \cup \text{Horn} \texttt{Horn} \text{(11)} \\
(\texttt{calloc}) & = x & (\texttt{calloc}L) & = \{1 + H = H ; H = x ; ((\texttt{malloc}L) \, \texttt{calloc} \, \texttt{malloc}) \, \texttt{havoc} \, \texttt{havoc} \, \texttt{assert} \, \texttt{assert} \, \text{Horn} \text{(01)} \\
(\texttt{store}) & = x & (\texttt{store}L) & = \{(1 + \phi \, H + 1) ; H = x ; ((\texttt{havoc}L) \, \texttt{havoc} \, \text{assert} \, \text{assert} \, \text{Horn}) \, \text{Horn} \text{(6)} \\
(\texttt{load}) & = x & (\texttt{load}L) & = \{(1 + \phi \, H + 1) ; H = x ; ((\texttt{havoc}L) \, \text{assert} \, \text{assert} \, \text{Horn}) \, \text{Horn} \text{(8)}
\end{align*}
TriCera is implemented in Scala, and it employs ELDARICA [6] as its Horn clause solver. Input programs first need to be parsed according to a grammar. Using a grammar for input C files, BNF Converter [36] is used to generate a compiler front end (i.e. a lexer, a parser and an abstract syntax definition) in Java, which in turn is used by TriCera.

Then, variable information is collected from their declarations and their types are determined, which can be one of the types given in Figure 9.1. Each statement is then translated into Horn clauses.

In TriCera, variables are classified as either local or global. Global variables exist in all generated clauses, and local variables only exist as long as they are in scope. Each variable has an associated term and an associated type, which is essentially symbolic execution. When unrolling the stack at a program location, the value of each variable is printed in the clause. The value is the associated term if the value is not updated, or the new value if it is updated.

E.g. consider the statement \( x = x + 42; \). Assume that the program has only one variable named \( x \), so the generated clause for this statement would be \( P_{\text{entry}}(x) \rightarrow P_{\text{exit}}(x + 42) \). Here \( x \) is the associated term. The type is not visible in Horn clauses, however, it is inferred from the input program while parsing (in this case int). The formula “\( x + 42 \)"
9.1 structs

structs are encoded using the support for theory of ADTs from ELDARICA. When TriCera encounters a struct definition for the first time, a new ADT for that struct is created. ADTs are encoded as functions with arity equal to the number of fields of the corresponding struct type. This allows the struct to be stored in a single variable. For the struct given in Listing 7.1, this would be the function: $S(f_1, f_2)$.

CCStruct is one of many types possible in TriCera, which all extend CCTYPE. Some of the other types which extend CCTYPE are arithmetic types and pointer types, as shown in Figure 9.1.

9.2 Stack Pointers

Stack pointers in TriCera are created when the "address of" operator (i.e. "&") is used on a stack variable or on the field of a stack variable. It extends the CCPointer type which is the base type for both stack and heap pointers.

Since structs are kept as ADTs, stack pointers to struct fields are not very straightforward to implement, since they cannot be directly found by looking at the list of global or local variables. For this reason, each stack pointer also has the field fieldAddress which is a list of integers that contains the address of the pointed struct field. If the list is empty (i.e. Nil), it means that the stack pointer is pointing to a regular variable on the stack. If the list has one element with value $n$, it means that the pointed value is the $n$th field of the pointed variable. If the list has more than one element, it means that the stack pointer is pointing to a nested struct.

9.3 Heap Pointers

The CCHeapPointer type contains a field, heapAlloc, for keeping information regarding the allocation site of the heap pointer and the corresponding invariant. The allocation site will be used for the refinements discussed in Chapter 8.1.3. The predicate is assumed while pulling, and asserted while pushing using the heap pointer.

The pull operation (i.e. load) creates a new variable on the stack, and any changes to this variable must be pushed back, thus a way to differentiate the pulled variables from regular variables is required. A special type, CCPulledVar, is used for this purpose which encapsulates the type of the value located on the heap.

9.4 Heap Operations

TriCera detects if the input program contains any heap allocation statements (i.e. malloc or calloc), and enables the heap support automatically by adding a global heap variable named "H". Since it is a global variable, it appears in every entry and exit predicate in the generated Horn clauses. $H$ is of type integer (unbounded), which is incremented when a new allocation is done, thus it simulates heap addresses in TriCera.

The pull and push operations are encapsulated using a Scala object named Heap. The Heap object keeps track of pulled variables and creates the required assertions and assumptions on pushes and pulls.
9.5 Parts Not Covered By This Thesis

TriCera supports a larger subset of the C language than described in this thesis, as explained in Section 1.2.2. Although not given as part of the grammar of CIMP, for loops, functions, C enum and programs with multiple threads are all supported by TriCera. Integers can also be modeled using mathematical (unbounded) or machine integer (32 bit or 64 bit) semantics.
Chapter 10

Testing and Experiments

This chapter first discusses the followed testing strategy while extending TriCera, and then gives information on the experiments carried out in order to evaluate the performance of the tool.

10.1 Testing

TriCera extends the functionality of ELDARICA’s Horn encoder for C programs, which means that the extensions should not break any other function. Regression tests are used for this purpose, and the initial un-extended implementation of TriCera / ELDARICA already contained a big selection of regression tests for the existing functionality.

With each extension to TriCera, relevant regression tests were added in order to decrease the chance of introducing undetected bugs and to make sure that the extensions work as expected. The following strategy was followed in creating the regression tests:

- test basic functionality and typical use cases,
- test tricky use cases and edge cases if applicable,
- there should be both tests which fail and generate a (correct) counter-example trace, and tests which succeed,
- add new regression tests for fixed bugs.

This strategy for creating new regression tests highlights the fact that writing and enhancing the set of tests cases is an iterative process, and the whole process depends on coming up with suitable use cases which requires experience. On the bright side, each added regression test ensures that the targeted bug will never go unnoticed again, increasing the quality of the tool. This is done by running the regression tests automatically at each update to the main repository of the tool.

Listing 10.1 shows an example regression test written to test the basic capability of stack pointers. It also tries to verify tricky cases possible in C, such as local variables shadowing global ones. Other regression tests for C structs and heap pointers were also added, which can be found at the main repository of the TriCera\textsuperscript{1}.

\textsuperscript{1}https://github.com/uuverifiers/tricera/tree/master/regression-tests
```c
int x;
int *y = &x;

void swap(int *x, int *y){
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

void main() {
    assert(*y == 0);  //global *y -> global x
    x = 1;
    assert(*y == 1);
    *y = 2;
    assert(x == 2);

    int a = 3;
    int b = 42;
    swap(&a, &b);
    assert(a == 42 && b == 3);

    int *x = &a, *y = &b;  //local *x and *y shadows the global ones
    swap(x, y);
    assert(a == 3 && b == 42);
    assert(*x == a && *y == b);
}
```

Listing 10.1: A unit test for stack pointers
10.2 Experiments

The experiments were carried out in order to evaluate the performance of TriCera, especially using benchmarks which have dynamic memory allocation, and compare the results against similar tools. For this purpose, 114 benchmarks from SV-COMP (Competition on Software Verification) were used. Benchmarks are chosen such that they allocate dynamic memory using `malloc` and do not contain any code currently unsupported by TriCera, such as arrays or pointer arithmetic. Names of the benchmark files are given in Appendix A.

The results are compared against the results obtained by running CPAchecker [37] on the same set of (preprocessed) benchmark files. CPAchecker was chosen as it was the most successful verification tool in SV-COMP 2019, for programs written in C [38]. Note that CPAchecker requires a configuration file for each benchmark, and the default property file provided with the tool is used instead of more specialized configuration files.

The benchmark files from SV-COMP include libraries and other files, and in some cases define macros. This makes it necessary to run the C preprocessor (cpp) on the input files before TriCera or CPAchecker can run the verification task on them.

To interpret the results, some definitions must be given.

- **safe**: The verification result produced by the tool is *safe* (i.e. solvable as shown in Figure 2.1)
- **unsafe**: The verification result produced by the tool is *unsafe* (i.e. unsolvable).
- **matched**: Each benchmark is expected to be either safe or unsafe, and the expected outcome is provided by SV-COMP. If the tool produced the expected verification result, it is a match. This is the total number of safe and unsafe matches.
- **imprecise**: The expected verification result is *safe*; however, the tool provided an *unsafe* result (i.e. a false alarm).
- **unknown**: The tool could not produce an outcome.
- **unsound**: The expected verification result is *unsafe*; however, the tool provided a *safe* result (a missed error).
- **error**: The tool cannot parse the input file or encountered an error during verification.

In addition to above definitions, in Table 10.1 and Table 10.2, *cpa* stands for CPAchecker version 1.8 from December 2018, *cpa-seq* stands for CPAchecker version 1.7.11 for SV-COMP 2019, and finally *tri* stands for TriCera version 0.1.

All experiments were ran on a system which has a total memory of 3021 MiB and has a Dual-Core AMD Opteron(tm) Processor 2220 SE. Java 1.8.0_121 was used during the experiments.

10.2.1 Results

The benchmarks are ran and results are collected first using a timeout of 1 minute (Table 10.1), and then using a timeout of 5 minutes (Table 10.2).

Some conclusions can be made looking at the results from Table 10.1 and Table 10.2.

- CPAchecker could verify more benchmarks.

---

2https://github.com/sosy-lab/sv-benchmarks/tree/master/c
### Table 10.1: Results - 1 minute timeout

<table>
<thead>
<tr>
<th></th>
<th>cpa</th>
<th>cpa-seq</th>
<th>tri</th>
</tr>
</thead>
<tbody>
<tr>
<td>matched</td>
<td>49</td>
<td>51</td>
<td>25</td>
</tr>
<tr>
<td>“</td>
<td>39</td>
<td>41</td>
<td>9</td>
</tr>
<tr>
<td>“</td>
<td>10</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>unsafe</td>
<td>1</td>
<td>1</td>
<td>85</td>
</tr>
<tr>
<td>unknown</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>error</td>
<td>64</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>114</td>
<td>114</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th></th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>30m26s</td>
<td>39m54s</td>
<td>12m57s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>73m0s</td>
<td>72m48s</td>
<td>47m2s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5m5s</td>
<td>4m46s</td>
<td>1m46s</td>
<td></td>
</tr>
</tbody>
</table>

### Table 10.2: Results - 5 minutes timeout

<table>
<thead>
<tr>
<th></th>
<th>cpa</th>
<th>cpa-seq</th>
<th>tri</th>
</tr>
</thead>
<tbody>
<tr>
<td>matched</td>
<td>53</td>
<td>51</td>
<td>25</td>
</tr>
<tr>
<td>“</td>
<td>39</td>
<td>41</td>
<td>9</td>
</tr>
<tr>
<td>“</td>
<td>14</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>unsafe</td>
<td>1</td>
<td>1</td>
<td>85</td>
</tr>
<tr>
<td>unknown</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>error</td>
<td>60</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>114</td>
<td>114</td>
<td>114</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th></th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>180m13s</td>
<td>183m36s</td>
<td>11m53s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>307m12s</td>
<td>304m19s</td>
<td>29m32s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13m4s</td>
<td>12m27s</td>
<td>0m50s</td>
<td></td>
</tr>
</tbody>
</table>
• CPAchecker produced more \textit{unknown} results, whereas TriCera produced more \textit{imprecise} results. CPAchecker produces an \textit{unknown} result instead of raising a false alarm when the verification result is uncertain (cannot conclude safe or unsafe) or if it cannot complete the analysis within the given timeout duration. TriCera produces an \textit{unsafe} result if it cannot verify the input to be safe; so in case of TriCera \textit{unknown} only indicates a timeout.

• TriCera has more \textit{unsafe} matches. This is most likely because TriCera simply says the benchmark is unsafe if it cannot find a solution (i.e. say that the benchmark is \textit{safe}), instead of declaring it unknown. This in turn leads to more \textit{unsafe} matches for TriCera.

• Examining the verified files closely, TriCera could verify 4 \textit{unsafe} benchmarks which CPAchecker could not verify. This is possibly due to the explanation given in the previous item.

• There were no unsound results produced by TriCera nor CPAchecker.

• TriCera is roughly 1.5 times faster for the 114 benchmarks when the timeout is set to 1 minute. When the timeout is set to 5 minutes, TriCera is roughly 10 times faster (user duration). This is because TriCera did not have any timeouts whereas most of the \textit{unknown} results for CPAchecker were due to timeouts.
Chapter 11

Conclusions

The goals of this project were to:

- add support for modeling structs as Horn clauses,
- find a method for encoding heap memory using Horn clauses,
- add support for heap memory and pointers (excluding pointer arithmetic),
- evaluate the performance of the implementation in comparison to other C model checkers.

The overall goal was to bring TriCera to a point where it can automatically model and verify C programs which contain pointers, heap memory interactions and structs.

The results given in Chapter 10 show that initial results for TriCera are promising. TriCera produced a reasonable amount of matches for the given benchmarks; however, it also produced a high number of false alarms. All used benchmarks contained pointers and heap memory interactions, and some of them contained structs.

The methodology of modeling heap, explained in Chapter 8, is similar to what was done in JayHorn; however, it is novel for a C model checker. The semantics of CIMP, while not fully covering the input language that TriCera accepts, is also given for the first time in Chapter 5.

11.1 Future Work

Stack pointers TriCera currently restricts stack pointers to be initialized at declaration, and to be non-reassignable as shown in Table 4.2. These constraints are in place as the current encoding strategy requires that the target of a stack pointer is unique. This limits the range of programs that TriCera can verify.

The limitations on stack pointers can be removed to cover a wider range of programs for verification. To remove the constraint that stack pointers cannot be reassigned requires points-to analysis to understand the possible variables that the stack pointer might still be pointing to, and this is a planned addition.

Heap refinements The refinements discussed in Chapter 8.1.3 would increase the precision of verification immensely. These improvements also require adding data-flow analysis. Data-flow analysis can be applied either during conversion to Horn clauses, or after the conversion directly on Horn clauses.
Bibliography


[34] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, “Verification of imperative programs through transformation of constraint logic programs.”, in VPT@CAV, 2013, pp. 30–41.


Appendices
Appendix A

Names of the SVCOMP’19 benchmark files used in the experiments

alternating_list-1.c  dll_nullified-2.c
alternating_list-2.c  dll-optional-1.c
calendar.c            dll-queue-1.c
cart.c                dll-rb-cnstr_1-1.c
c11_search-alloca-WithForgottenReturn_false-valid-deref.c
dll-01-2.c            dll-rb-sentinel-2.c
dll2c_append_equal.c  dll-reverse.c
dll2c_append_unequal.c dll-simple-white-blue-2.c
dll2c_insert_equal.c  dll-sorted-2.c
dll2c_insert_unequal.c dll-token-1.c
hash_fun.c
list-1.c
list-2.c
list-ext_1.c
list-ext.c
list-ext_flag_1.c
list-ext_flag.c
list_flag-1.c
list_flag-2.c
list_search-1.c
list_search-2.c
lockfree-3.0.c
min_max.c
packet_filter.c
quick_sort_split.c
running_example.c
shared_mem1.c
shared_mem2.c
simple-1.c
simple-2.c
simple_and_skiplist_2lvl-1.c
simple_and_skiplist_2lvl-2.c
| simple_built_from_end.c          | sll2n_update_all_reverse.c        |
| simple-ext_1.c                 | sll-buckets-1.c                  |
| simple_search_value-1.c        | sll-circular-1.c                 |
| simple_search_value-2.c        | sll_circular_traversal-1.c       |
| skiplist_2lvl.c                | sll_circular_traversal-2.c       |
| skiplist_3lvl.c                | sll_length_check-1.c             |
| sll-01-1.c                     | sll_length_check-2.c             |
| sll2c_append_equal.c           | sll_nondet_insert-1.c            |
| sll2c_append_unequal.c         | sll_nondet_insert-2.c            |
| sll2c_insert_equal.c           | sll_of_sll_nondet_append-1.c     |
| sll2c_insert_unequal.c         | sll_of_sll_nondet_append-2.c     |
| sll2c_prepend_equal.c          | sll_optional-1.c                 |
| sll2c_prepend_unequal.c        | sll-optimal-1.c                  |
| sll2c_remove_all.c             | sll-queue-1.c                    |
| sll2c_remove_all_reverse.c     | sll-rb-cnstr_1-1.c               |
| sll2c_update_all.c             | sll-rb-sentinel-1.c              |
| sll2c_update_all_reverse.c     | sll-reverse_simple.c             |
| sll2n_append_equal.c           | sll_shallow_copy-1.c             |
| sll2n_append_unequal.c         | sll_simple-white-blue-1.c        |
| sll2n_insert_equal.c           | sll-sorted-2.c                   |
| sll2n_insert_unequal.c         | sll-token-2.c                    |
| sll2n_prepend_equal.c          | splice-1.c                       |
| sll2n_prepend_unequal.c        | splice-2.c                       |
| sll2n_remove_all.c             | test-0504_1.c                    |
| sll2n_remove_all_reverse.c     | test-0504.c                      |
| sll2n_update_all.c             | tree_of_cslls.c                  |