Automated Inference of ACSL Contracts for Programs with Heaps

OSKAR SÖDERBERG
Abstract

Contract inference consists in automatically computing contracts that formally describe the behaviour of program functions. Contracts are used in deductive verification, which is a method for verifying whether a system behaves according to a provided specification. The SAIDA plugin in Frama-C is a contract inference tool for C code. This thesis explores an extension to the SAIDA plugin which allows support for pointers and heap allocations. The goal is to evaluate to what extent model checking tools can be used to infer contracts for deductive verification of programs that use pointers and heap allocations. This is done by proposing a translation strategy to convert contracts containing heap expressions, generated by the model checker TriCera, into ACSL, a specification language used by Frama-C. An implementation of this translation is evaluated using a set of verification tasks. The results demonstrate that the inferred contracts are sufficient to verify simple code samples, including features such as recursion, aliasing, and manipulation of heap-allocated structs. However, the results also reveal cases where the contracts are too weak, although more information could be extracted in the translation. It is concluded that model checking tools can infer contracts for deductive verification of programs with pointers and heap allocations, but currently to a limited extent. Several improvements to the translation strategy are proposed for future work.

Keywords

Formal Verification, Contract Inference, Model Checking, Deductive Verification, Theory of Heaps, ACSL, Translation
Sammanfattning


Nyckelord

Formell Verifiering, Kontrakthärledning, Modellprovning, Deduktiv Verifiering, Theory of Heaps, ACSL, Översättning
Acknowledgements

I would like to thank my supervisors Christian Lidström and Gustav Ung for their support throughout the project, for the knowledge shared, and for giving valuable feedback. I would also like to thank Philipp Rümmer and Zafer Esen for many insightful meetings and conversations.
Contents

1 Introduction 6
  1.1 Problem Statement 7
  1.2 Purpose 7
  1.3 Ethical Considerations 7

2 Background 8
  2.1 Deductive Verification 8
    2.1.1 The While Language 8
    2.1.2 Hoare Logic 8
    2.1.3 Weakest preconditions 12
    2.1.4 Procedures and Modularity 14
    2.1.5 Frama-C and the WP plugin 14
    2.1.6 ACSL 15
    2.1.7 The Verification Process 16
  2.2 Model Checking 17
    2.2.1 The Extended While Language 17
    2.2.2 Constrained Horn Clauses 17
    2.2.3 Modelling Programs using Constrained Horn Clauses 18
    2.2.4 The SMT-LIB Standard 21
    2.2.5 Theory of Heaps 22
  2.3 Contract Inference 24
    2.3.1 TriCera 24
    2.3.2 Saida 25
  2.4 Related Work 25
    2.4.1 Automatic Contract Inference 25
    2.4.2 Reasoning with Pointers and Heaps 28

3 Translating Theory of Heap Contracts 28
  3.1 Implicit vs Explicit Heap Expressions 28
  3.2 Translation Development 29
  3.3 The Translation Strategy 29
    3.3.1 Overview 30
    3.3.2 PostconditionSimplifier 30
    3.3.3 PointerPropExtractor 31
    3.3.4 AssignmentExtractor 32
    3.3.5 TOHProcessor 34
    3.3.6 ADTProcessor 35
    3.3.7 ToVarForm 35
    3.3.8 ACSLProcessor 36
    3.3.9 ClauseRemover 37
    3.3.10 ACSLLineariser 38
  3.4 Implementation 39
# 4 Evaluation

4.1 Verification Tasks ................................................. 40
4.2 Contract Inference .................................................. 41
4.3 Post-Processing ...................................................... 41
4.4 Verification Attempt ................................................. 41

# 5 Results

5.1 Discussion .......................................................... 43

# 6 Conclusion

6.1 Limitations ......................................................... 45
6.2 Future Work ........................................................ 45
6.2.1 Improved ClauseRemover ....................................... 46
6.2.2 Assigns Clauses .................................................. 46
6.2.3 Extracting More Validity and Separateness Information .. 46
6.2.4 A More General Heap Extractor ............................... 46
6.2.5 Further Testing .................................................. 46

References ................................................................. 50

# A Appendix: Code Samples

A.1 get-1.c .............................................................. 51
A.2 get-2.c .............................................................. 52
A.3 incdec-1.c ........................................................... 53
A.4 incdec-2.c ........................................................... 54
A.5 incdec-3.c ........................................................... 55
A.6 max-1.c .............................................................. 56
A.7 max-2.c .............................................................. 57
A.8 multadd-1.c ........................................................ 58
A.9 multadd-2.c ........................................................ 59
A.10 multadd-3.c ......................................................... 61
A.11 truck-1.c .......................................................... 63
A.12 truck-2.c .......................................................... 64
1 Introduction

Safety-critical systems, as found for example in cars, aeroplanes, and the nuclear industry, may cause dire consequences upon failure. Thus, it is of interest to develop techniques that minimise the risk of unexpected behaviour in such systems. The area of formal verification aims to solve this by describing systems mathematically and verifying their correctness. With such techniques, it is possible to build systems with very high safety guarantees and thereby prevent mishaps.

Frama-C with its WP plugin is a tool for formal verification of C code which builds on a technique called deductive verification, where a code base is verified step by step in a modular fashion. It is usually required that a verification engineer provides code annotations to guide the tool in generating a proof, which can be a challenging and time-consuming task. An alternative to formal verification is extensive testing, which requires constructing a large variety of tests and running those on the target system. This approach also requires much work from the developers. The challenge that safety-critical system development imposes is exemplified in [22] where it is stated that a significant part of the R&D employees at the heavy vehicles manufacturer Scania develops embedded software, out of which most parts are safety-critical. However, if the process of formally verifying code got faster, it could save companies like Scania a significant amount of work spent on system development and increase system safety.

In this spirit, a new method for automating parts of the verification process was recently proposed in [3]. The new method combines deductive verification with model checking, where the verification is attempted on the program as a whole. In the process of verifying a program, model checkers can also infer function specifications which can be used by Frama-C, and would otherwise need to be written by hand. By combining deductive verification with model checking to automatically infer specifications, the amount of manual work needed for verification is reduced. The plugin Saida makes this method easily accessible in Frama-C by providing the necessary translation of specifications and inferring new specifications using the model checker TriCera. However, there are multiple ways to improve the method, including both fundamental features, like handling pointers and heap allocations, and progressive features, like supporting contracts with uninterpreted predicates. These improvements and more are discussed in [5].

This thesis will aim to investigate how this contract inference method can be extended to handle pointers and heap allocations. In its current state, TriCera supports pointers and heap allocations through the theory of heaps as presented in [14], which formally models heaps and interactions with them. However, there is currently no translation model from formulas in the theory of heaps to annotations expressed in the specification language ACSL that can be used by the deductive verifier Frama-C. Thus this gap needs to be filled.
1.1 Problem Statement

The goal of this thesis is to extend the SAIDA plugin with support for pointers and heaps, in order to evaluate the method of inferring annotations for the verification of programs with pointers and heap allocations. This is formulated in the main research question of the thesis:

To what extent can model checking tools be used to infer contracts for deductive verification of programs using pointers and heap allocations?

This question can be divided into three more specific subquestions:

1. How can formulas involving the theory of heaps, as presented in [14], be translated into ACSL annotations that can be used for deductive verification?

2. How does the SAIDA plugin, extended to support pointers and heap allocations, perform on a collection of verification tasks, in terms of what types of programs can be verified?

3. What are the limitations of the proposed translation strategy from formulas involving the theory of heaps to ACSL annotations?

To answer these questions, the translation model in subquestion 1 will be implemented into TRICERA, and then the SAIDA plugin will be tested on a set of verification tasks that include pointers and heap allocations.

1.2 Purpose

The purpose of the project is to evaluate the potential for contract inference in a broader context compared to what has been done before. This provides insight into the potential of the technique and its limitations which helps guide further research in the area. Looking at the bigger picture, continued research on formal verification can lead to more robust systems at a cheaper cost, saving lives and resources.

1.3 Ethical Considerations

As formal verification may be used to verify safety-critical systems where malfunction can cause dire consequences, it is important that work within this area is carried out with great care to avoid faulty guarantees being made. For this reason, it is important to reason about the correctness of work in this area. For the method used in this thesis, the safety guarantees are made by the Frama-C WP plugin (see Section 2.1.5) which attempts to prove that each provided specification holds. It follows that the inferred specifications cannot cause false correctness guarantees, as they are checked by the prover.

Another ethical aspect is that automation may result in people losing their jobs. On the other hand, by automating verification, more time can be spent on building better systems that can improve the lives of many. Additionally, new jobs in the form of formal verification engineers could be created.
2 Background

This section presents the techniques behind contract inference and the necessary background to understand an extension to support pointers and heaps. Subsections 2.1 and 2.2 cover two verification techniques, where the latter is employed to infer contracts, which are then utilised by the former technique. These sections include introductions to the specification language ACSL and to the theory of heaps, which provide different ways of expressing pointers and heaps and are central to extending the contract inference capabilities of the technique.

2.1 Deductive Verification

In deductive verification, the correctness of a program is validated by transforming the program into a number of proof obligations, which can be verified independently. This section covers the underlying theory for this approach to verification. We will start by presenting a simple programming language called the while language which allows us to focus on the central concepts and omit irrelevant details for this thesis. In the same spirit, we will let our program variables be capable of storing integers of arbitrary size. The ideas presented here extend to C, but this requires a more careful treatment.

2.1.1 The While Language

We define a simple programming language denoted the while language, consisting of assignments, if-statements and while loops inspired by [21].

Definition 2.1. The syntax of a statement, or a program, in the while language is defined as

\[ S ::= S_1 ; S_2 | x := a | \text{if } b \text{ then } S_1 \text{ else } S_2 | \text{while } b \text{ do } S_1 | \text{skip} \]

where \( S_1, S_2 \) are statements, \( x \) is a program variable, \( a \) is an integer expression and \( b \) is a boolean expression.

2.1.2 Hoare Logic

Hoare logic [17] defines a foundation for reasoning about programs and their correctness. A fundamental building block of deductive verification is using Hoare triples and contracts [20] to describe the properties of a function.

Definition 2.2. Let \( P \) and \( Q \) be assertions, i.e. formulas of predicate logic, and \( S \) be a program.

- We denote by \( \{P\}S\{Q\} \) a Hoare triple of a program \( S \), with precondition \( P \) and postcondition \( Q \).
- We denote by \( C = (P, Q) \) a contract representing a pair of a precondition \( P \) and a postcondition \( Q \).
This notation is used to formally describe the effect of a program and to define program correctness. We note that a contract is an abstraction of a Hoare triple allowing us to reason about pairs of preconditions and postconditions independently of programs. We will now define the correctness notion of a Hoare triple as in §2 and work our way towards defining the satisfaction notion of a contract.

**Definition 2.3.** Let \( s \) denote a state, i.e. a mapping from program variables to values. We say that \( \{P\}S\{Q\} \) holds with respect to partial correctness if executing \( S \) from any state \( s \) where \( P \) holds, either diverges or terminates in a state \( s' \) where \( Q \) holds.

Similarly, we say that \( \{P\}S\{Q\} \) holds with respect to total correctness if executing \( S \) from any state \( s \) where \( P \) holds, terminates in a state \( s' \) where \( Q \) holds.

We note that a Hoare triple that holds with respect to total correctness also holds with respect to partial correctness.

**Example 2.1.** Let us denote Listing 1 from §3 by \( \text{set} \, x \), which sets the value of \( x \) equal to \( n \) given that \( n \geq 0 \). We can describe these properties of the program using the Hoare triple \( \{n \geq 0\} \text{set} \, x \{x = n\} \), or independently of the program using the contract \( (n \geq 0, x = n) \). In fact, \( \{n \geq 0\} \text{set} \, x \{x = n\} \) holds with respect to partial and total correctness, and we will show the former in Example 2.2.

```
Listing 1 set x
1 x := 0;
2 while (x < n) do
   x := x + 1
```

Now, using Hoare triples, we can define proof rules which allow us to step-by-step build a proof for the correctness of a program. To this end, we consider the Hoare rules in Definition 2.4 as presented in §2 (but first introduced in §7).

We recall that \( \frac{A}{C} \frac{B}{C} \) denotes that from the two formulas above the bar, the formula below the bar can be derived. Further, \( P[E/x] \) denotes formula \( P \) with all free occurrences of \( x \) replaced by \( E \).

**Definition 2.4.** We define the following Hoare rules:

- **Composition**
  \[
  \frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1 \; ; \; S_2\{R\}}
  \]

- **Assignment**
  \[
  \frac{}{\{R[a/x]\}x := a\{R\}}
  \]

- **Condition**
  \[
  \frac{\{P \land B\}S_1\{R\} \quad \{P \land \neg B\}S_2\{R\}}{\{P\}\text{if } B \text{ then } S_1 \text{ else } S_2\{R\}}
  \]

9
We can use the consequence rule to derive the two alternative forms presented in Definition 2.4. These versions of the Hoare rules facilitate the construction of proofs as we will see in Example 2.2.

Definition 2.5. We additionally have:

\[
\begin{align*}
\text{Assignment'}: & \quad P \implies R[a/x] \\
\text{LOOP'}: & \quad P \implies I \quad (I \land B) S[I] \quad I \land \neg B \implies R \\
\end{align*}
\]

To see how the rules can be used to formally verify a simple program, we consider Example 2.2.

Example 2.2. Consider the Hoare triple from Example 2.1

\[
\{n \geq 0\} x := 0 \ ; \ \text{while} \ (x < n) \ \text{do} \ x := x + 1 \{x = n\}.
\]

We want to verify that this Hoare triple holds. By using the Hoare rules in Definition 2.4 and 2.5, we construct the following proof tree starting from the bottom and working our way up.

\[
\begin{align*}
& n \geq 0 \implies P[0/x] \\
& P \implies I \\
& I \land x < n \implies I[x + 1] \\
& \{n \geq 0\} x := 0 \{P\} \\
& \{P\} \text{while} (x < n) \ \text{do} \ x := x + 1 \{x = n\}
\end{align*}
\]

Note that we had to introduce two new predicates \(P\) and \(I\). In order to close the proof, it remains to find \(P\) and \(I\) so that all of the top level formulas hold:

\[
\begin{align*}
& n \geq 0 \implies P[0/x] \\
& P \implies I \\
& I \land x < n \implies I[x + 1] \\
& I \land \neg(x < n) \implies x = n
\end{align*}
\]

One possible solution is \(P \triangleq x \leq n\) and \(I \triangleq x \leq n\). Thus, when we execute the program from any state \(s\) that satisfies the precondition \(n \geq 0\), the final state will satisfy \(x = n\), if the program terminates. Thus, the Hoare triple holds with respect to partial correctness.
We have now seen how the correctness of a program can be shown. Note however, that the Hoare triple we considered in Example 2.2 does not state the exact values of \( x \) and \( n \) after the program, only the fact that they are equal. In fact, they will be equal to the value of \( n \) before the program was executed. To be able to express properties in terms of initial conditions, we introduce logical variables and interpretations inspired by [5].

**Definition 2.6.** A logical variable is a variable that does not occur in a program and is therefore not altered during program execution. Let the set of logical variables in a program be denoted by \( L \). We define an interpretation \( I \) to be a mapping from logical variables to values, which here is a mapping \( L \to \mathbb{Z} \). When stating whether a Hoare triple with logical variables holds, we do so with respect to an interpretation \( I \), where the values of the logical variables are given by \( I \).

We will denote logical variables with a 0-subscript, for example, \( x_0 \). Let us consider an example to show how logical variables can be used to express a Hoare triple.

**Example 2.3.** Again, consider the program \texttt{set \( x \)} from Example 2.1. Let \( x \) and \( n \) be variables, and \( n_0 \) be a logical variable. We can now more accurately capture the final values of \( x \) and \( n \) with the Hoare triple \[
\{ n = n_0 \land n \geq 0 \} \texttt{set \( x \)} \{ x = n \land n = n_0 \}
\]
or the corresponding contract \[(n = n_0 \land n \geq 0, x = n \land n = n_0).\]

These notions state that if \( n \geq 0 \) and \( n \) is equal to the initial value \( n_0 \) before program execution, then \( x \) and \( n \) are equal to the initial value \( n_0 \) after program execution. This rules out the possibility of modifying \( n \) in order to satisfy the contract, which was allowed in the version of the contract from Example 2.1. The Hoare triple holds with respect to partial correctness and any interpretation of logical variables. The proof is left as an exercise for the reader.

We are now ready to define what it means for a program to satisfy a contract. The definition is inspired by [5].

**Definition 2.7.** Let \( I \) be an interpretation. We say that a program \( S \) satisfies a contract \( C \) with respect to \( I \), denoted by \( S \models_I C \), if and only if \( \{ P \} S \{ Q \} \) holds, where the values of the logical variables are given by \( I \). We further say that a program \( S \) satisfies a contract \( C \), denoted by \( S \models C \), if and only if \( \forall I. S \models_I C \). To specify the correctness notion, we use \( \models^p \) for partial correctness and \( \models^t \) for total correctness.

To get a sense of how the satisfies notion is used, we consider Example 2.4.
Example 2.4. Denote the contract corresponding to the Hoare triple in Example 2.3 by $C \triangleq (n = n_0 \land n \geq 0, x = n \land n = n_0)$. Since the Hoare triple holds with respect to partial correctness and every interpretation, $\text{set}_x \models^p C$. Now, considering Listing 2, it is clear that $\text{zero} \not\models C$. However, $\text{zero} \models^I C$ with respect to the interpretation $I \triangleq [n_0 \mapsto 0]$.

2.1.3 Weakest preconditions

Let us now consider the problem of verifying whether a program satisfies a contract. Example 2.2 showed how this can be done by building a proof step-by-step and finally showing that we can find predicates that satisfy the top-level conditions. In this section, we will further refine this method by reasoning about weakest preconditions. The idea is that given a contract $(P, Q)$ and a program $S$, we will consider all states (including logical variables) for which executing $S$ results in a new state that satisfies $Q$. We then show that any state that satisfies $P$ is among these states. We start by defining two types of weakest preconditions, inspired by [10].

Definition 2.8. Given a program $S$ and a postcondition $Q$, the weakest liberal precondition is an assertion $P_{wl}$ which holds exactly for the states and interpretations such that executing $S$ either terminates in a state satisfying $Q$ or diverges. The weakest conservative precondition is an assertion $P_{wc}$ which holds exactly for the states and interpretations such that executing $S$ terminates in a state satisfying $Q$.

Weakest preconditions express the weakest assertions that must be satisfied for a program execution to end in a state satisfying the postcondition (or diverge in the case of liberal preconditions). As a foundation for reasoning with weakest preconditions, we present Theorem 2.1.

Theorem 2.1. Let $S$ be a program, $(P, Q)$ a contract, and $P_{wl}$ the weakest liberal precondition with respect to $S$ and $Q$. Then

$$S \models^p (P, Q) \iff (P \implies P_{wl}).$$

For the weakest conservative precondition $P_{wc}$ with respect to $S$ and $Q$, we have

$$S \models^t (P, Q) \iff (P \implies P_{wc}).$$

Proof. We show the statement for the weakest liberal precondition. We start by showing $\implies$. Assume that $S \models^p (P, Q)$. Then $(P)S(Q)$ holds with respect to
partial correctness and for every interpretation. Thus, \( P \) holds only for states and interpretations such that executing \( S \) either diverges or terminates in a state where \( Q \) holds. Thus, by definition, \( P \implies P_{\text{w}} \). We now show the other direction, \( \iff \). Assume that \( P \implies P_{\text{w}} \). By definition, \( \{P_{\text{w}}\}S\{Q\} \) holds with respect to partial correctness and every interpretation. But since \( P \implies P_{\text{w}} \), executing \( S \) from any state and interpretation satisfying \( P \) either diverges or terminates in a state where \( Q \) holds. So \( S \models (P, Q) \).

With the help of Theorem 2.1, the task of verifying that a program \( S \) satisfies a given contract \((P, Q)\) can be divided into first calculating the weakest precondition \( P_{\text{w}} \), and then showing that \( P = P_{\text{w}} \). Computing the weakest precondition can easily be done for the while language if we exclude while-statements and assume termination, using the weakest precondition calculus introduced by Dijkstra \[10\]. The calculus is presented in Definition 2.9 with a slightly modified treatment of if-statements for clarity.

**Definition 2.9.** We define the \( wp \)-function:

- \( wp(\text{skip}, Q) \triangleq Q \)
- \( wp(x := a, Q) \triangleq Q[a/x] \)
- \( wp(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) \triangleq \)
  \( (b \implies wp(S_1, Q)) \land (\neg b \implies wp(S_2, Q)) \)
- \( wp(S_1 ; S_2, Q) \triangleq wp(S_1, wp(S_2, Q)) \)

Given a postcondition \( Q \) and a program \( S \), the weakest precondition is given by \( P_{\text{w}} = wp(S, Q) \). Note that the liberal and conservative notions are equivalent in this context as termination is guaranteed. To see how the \( wp \)-function is used, we consider Example 2.5.

**Example 2.5.** Consider Listing 3 which we denote by \( \text{max} \). We will show that the program satisfies the contract \((P, Q)\), where

\[
(P, Q) \triangleq (x = x_0 \land y = y_0, \ (x_0 < y_0 \implies r = y_0) \land (x_0 \geq y_0 \implies r = x_0)).
\]

The computation of the weakest precondition proceeds as follows:

\[
P_{\text{w}} = wp(\text{if } x < y \text{ then } r := y \text{ else } r := x, Q)
\]

\[
= (x < y \implies wp(x := y, Q)) \land (x \not< y \implies wp(r := x, Q))
\]

\[
= (x < y \implies Q[y/r]) \land (x \not< y \implies Q[x/r])
\]

\[
= (x < y \implies ((x_0 < y_0 \implies y = y_0) \land (x_0 \geq y_0 \implies y = x_0)))
\]

\[
\land (x \not< y \implies ((x_0 < y_0 \implies x = y_0) \land (x_0 \geq y_0 \implies x = x_0)))
\]

Now, we want to show that \( P \implies P_{\text{w}} \). Assume that \( x_0 < y_0 \). Then we have \( x < y \) and \( y = y_0 \) by \( P \), so \( P_{\text{w}} \) holds. Now assume \( x_0 \geq y_0 \). Then we have \( x \geq y \) and \( x = x_0 \) by \( P \), so \( P_{\text{w}} \) holds in this case as well. Since \( P \implies P_{\text{w}} \), we have that \( S \models (P, Q) \) by Theorem 2.1.

\[\square\]
Calculating the weakest liberal preconditions in the presence of while-loops is more complicated as the number of iterations a loop performs is not always known. In fact, weakest preconditions are not always computable. One approach to this problem is to relax the requirement of finding the weakest precondition and instead attempt to find some stronger, but yet weak enough liberal precondition or conservative precondition to show that a program satisfies a contract. Based on the previously defined weakest preconditions, we have Definition 2.10.

**Definition 2.10.** Let $S$ be a program, $Q$ be a postcondition, and $P_{wl}$, $P_{wc}$ be the corresponding weakest liberal and conservative preconditions. A **liberal precondition** is an assertion $P_l$ such that $P_l \implies P_{wl}$ and a **conservative precondition** is an assertion $P_c$ such that $P_c \implies P_{wc}$.

If we have a precondition $P$ for some program $S$ and postcondition $Q$, we need only show that $P \implies P_l$ to know that $S \models^P (P, Q)$ (by Theorem 2.1 using that $P_l \implies P_{wl}$). The positive side of this relaxation is that it may be possible to compute these preconditions for programs with loops. On the negative side, if $P_l$ is too strong, it may hold that $P \implies P_{wl}$ but not $P \implies P_l$, making a proof impossible to conclude by this strategy. These ideas are important to understanding the strengths and weaknesses of verification tools based on weakest preconditions.

### 2.1.4 Procedures and Modularity

A fundamental programming language feature is procedure calls. The way deductive verification methods handle procedures is at the heart of the modularity these methods provide. Suppose that we have a program `main` that calls the procedures `set_x` and `max` as part of the program. By providing contracts for the three procedures, we can show that `main` satisfies its contract under the assumption that `set_x` and `max` satisfy their contracts. By then showing that `set_x` and `max` satisfy their contracts, the verification of the `main` program is completed. This method can be generalised to code bases of any size and allows us to verify one procedure at a time, and reuse functions and their contracts. A rigorous treatment of procedure calls in Hoare logic is presented in [23].

### 2.1.5 Frama-C and the WP plugin

**Frama-C** [19] is a platform for analysis of C code, specifically aimed at showing correctness properties of code. Different verification methods are implemented.
in plugins which operate on a common kernel and use the same formal specification language ACSL [7], which will be introduced in Section 2.1.6. This common interface lays a robust foundation for building additional Frama-C extensions which may build on previous implementations. This thesis will utilise the WP plugin [6], which is a verification tool based on the ideas of the weakest precondition calculus introduced in Section 2.1.3. It should be noted, however, that much complexity is added when going from the while language to C, one example being the treatment of aliased pointers. In practice, specifications and annotations such as procedure contracts, loop invariants and assertions must be provided to guide the verification process, which are provided using ACSL.

Although the WP plugin does support dynamic memory allocation, one current limitation is that it does not support specifying allocations and deallocations in contracts. For this reason, the procedures we infer contracts for in this thesis will be prohibited from using allocations and deallocations as the contracts cannot accurately describe this behaviour.

### 2.1.6 ACSL

In order to express contracts and other code annotations, Frama-C uses ACSL (ANSI/ISO C Specification Language) [7]. This specification language supports expressions corresponding to pure C expressions and additional important constructs such as quantifiers. Some example syntax for common constructs include:

- `<` (less than), `<=` (less than or equals), `==` (equals), `&&` (and), `||` (or), `=>` (implication)
- `\forall` (universal quantification), `\exists` (existential quantification)

ACSL specifications and annotations are written in the source code inside a comment starting with an `@`-sign. A contract in the specification language consists of `requires` clauses whose conjunction defines the precondition, and `ensures` clauses whose conjunction defines the postcondition. Additionally, `assigns` clauses may be used to declare the frame condition, i.e. what memory locations are modified by the program. This is needed to specify that the variables not mentioned in the contract should remain unchanged. If no location is modified, this can be specified with the `\nothing` construct. When referring to the value of a variable before the program execution, the `\old()` construct can be used, analogous to the use of logical variables in Section 2.1.2. When referring to the returned value of a program, the `\result` construct is used. Constructs for handling pointers are available in ACSL and supported by the WP plugin. Let `p` be a pointer and `s1, s2` be sets of pointers. Then

- `\valid(p)` returns `true` if `*p` is guaranteed to produce a definite value according to the C standard and `false` otherwise.
- `\separated(s1, s2)` returns `true` if all locations in `s1` and `s2` are disjoint, and `false` otherwise.
An example contract written in ACSL is given in Listing 4.

**Example 2.6.** Consider the C function and ACSL contract presented in Listing 4. This contract states as precondition that the input pointers \(x\) and \(y\) are valid pointers, and states as postcondition that if \(*x < *y\) then the returned value will be the initial value of \(*y\). Similarly, if \(!(*x < *y)\) then the initial value of \(*x\) is returned. Finally, it is stated that no memory locations are modified in the `assigns` statement.

```
/*@ requires \valid(x) && \valid(y);
  ensures *x < *y ==> \result == \old(*y);
  ensures !(*x < *y) ==> \result == \old(*x);
  assigns \nothing;
*/

int max_pointer(int* x, int* y) {
  if (*x < *y) {
    return *y;
  } else {
    return *x;
  }
}
```

While ACSL includes the keywords `allocates` and `frees` for specifying the allocation and deallocation of memory, these are not currently supported by the WP plugin. Thus, inference of contracts for procedures using allocation and deallocation of memory will not be considered in this thesis.

### 2.1.7 The Verification Process

When verifying a code base using the WP plugin in FRAMA-C, the verification process can be divided into two parts. First, a formal correctness specification of the main program needs to be written in the form of a contract, which we will denote as the *main contract*. This contract defines some arbitrary properties we ultimately want to show that the program satisfies. The second part consists in using the WP plugin to verify that the main program satisfies the main contract. As the WP plugin uses callee contracts to verify callers in the way that was discussed in Section 2.1.4, we must write contracts for each procedure potentially called by the program, and provide annotations including loop invariants and assertions that are sufficient for the plugin to verify that each procedure satisfies its contract. In this thesis, we will consider contract inference to aid in the second part of the verification process.
2.2 Model Checking

Model checking consists of verifying whether a model of a system fulfills some property. Here we will consider a method where programs are translated into a set of constrained Horn Clauses, on which a satisfiability check can be performed by a Horn solver. The idea is to translate the program in such a way that the satisfiability test provides some guarantee about the properties of the program.

2.2.1 The Extended While Language

We will consider the while language introduced in Section 2.1.1 extended with the statements

\[ \text{assume } b \mid \text{assert } b \]

where \( b \) is a boolean expression. The \text{assume} statement blocks execution indefinitely if \( b \) evaluates to \text{false} and the \text{assert} statement raises an error if \( b \) evaluates to \text{false}. Otherwise, the semantics of the two statements is equivalent to \text{skip}. These new statements will be used to specify program requirements, similar to the role contracts played for talking about program correctness in Section 2.1.2.

2.2.2 Constrained Horn Clauses

A constrained Horn clause (CHC) is a special form of logical formula which can be used to encode programs. By expressing verification problems in this form, the verification task is given a common interface on which different CHC-solvers can search for a solution. Before introducing CHCs, we define some useful notation.

Definition 2.11. We define

- A term is a constant or a variable, or an application of a function to a term.
- A predicate is a function that evaluates to true or false.
- A clause is a disjunction of predicates, or their negations, where the variables are implicitly universally quantified.

Example 2.7. The following are examples of how the notation in Definition 2.11 can be used in practice.

- Let \( x \) and \( y \) be variables. Then \( x, y, 42 \) and \( x + 1 \) are examples of terms.
- Define \( \text{pos}(z) \triangleq (z > 0) \) and \( \text{odd}(z) \triangleq (z \equiv 1 \mod 2) \). Then \( \text{pos} \) and \( \text{odd} \) are predicates.
- An example of a clause is \( (\text{pos}(x + 1) \lor \neg\text{odd}(42)) \), which is equivalent to the statement \( \forall x. (\text{pos}(x + 1) \lor \neg\text{odd}(42)) \).
One last notion needs to be covered before we are ready to define CHCs, which is the concept of a background theory. Such a theory formalises certain logical facts about a construct and thereby allows us to reason about it. Examples include the theory of Linear Arithmetic, which allow us to express linear formulas and (in)equalities between these, or the theory of Arrays which allow us to reason about reads and writes to arrays. In Section 2.2.5, we will describe such a theory for reasoning about heaps. We are now ready to define CHCs based on the definition in [28]:

**Definition 2.12.** A constrained Horn clause is a sentence

\[ \forall x_1, x_2, \ldots (C \land B_1 \land \ldots \land B_n \rightarrow H) \]

where

- \( H \) is either false or an application of a \( k \)-ary predicate \( p(t_1, \ldots, t_k) \) to first-order terms.
- Each \( B_i \) is an application of an \( m \)-ary predicate \( p(t_1, \ldots, t_m) \) to first-order terms.
- \( C \) is a constraint over some background theories.

We note that outside this definition, the universal quantification over variables will be implicit due to our definition of a clause.

Further, we define a solution to a set of Horn clauses as a mapping from the predicates of the clauses to constraints over their arguments, such that replacing predicates by their corresponding constraints makes all clauses valid.

**Example 2.8.** Let \( p_1 \) and \( p_2 \) be predicates. The formula

\[ x \geq 0 \land p_1(x) \rightarrow p_2(x) \]

is an example of a CHC where \( C \triangleq x \geq 0 \), \( B_1 \triangleq p_1(x) \) and \( H \triangleq p_2(x) \). A possible solution is \( p_1(x) \triangleq \text{true} \) and \( p_2(x) \triangleq x \geq 0 \).

### 2.2.3 Modelling Programs using Constrained Horn Clauses

We will now see how programs can be modelled using CHCs. We start by viewing programs as transition systems, defined as in [12].

**Definition 2.13.** A transition system is a tuple \( (S, I, \rightarrow) \) where

- \( S \) is the set of states
- \( I \subseteq S \) is the set of initial states
- \( \rightarrow \subseteq S \times S \) is the transition relation, i.e the set of state transitions
Now, by denoting the set of program control locations by \( \text{Loc} \), and the set of mappings from program variables to values \( \text{Val} \), we can model a program state as a tuple \( (l, v) \in \text{Loc} \times \text{Val} = S \). This leads us to model the set of initial states as \( I = \text{Loc}_{\text{init}} \times \text{Val}_{\text{init}} \), where \( \text{Loc}_{\text{init}} \) is the set of initial control locations and \( \text{Val}_{\text{init}} \) is the set of initial variable-value mappings. Lastly, \( \to \) is modelled according to the effect that executing a single step from the control location has on the program state. Here, we will consider an example of how this is done, whereas a formal description is provided in \([12]\). Using transition systems, it is natural to visualise programs with \textit{program graphs}, where nodes correspond to states and edges correspond to state transitions. We will see how a transition system and a program graph can be constructed in the following example.

**Example 2.9.** Consider Listing \( \text{5} \), which is a C version of the previous \texttt{set\_x} program. The assume and assert statements specify the program properties we want to show, i.e. that the program assigns the value of \( n \) to \( x \), provided that \( n \geq 0 \). Let us denote mappings from program variables to values by tuples \((\text{val}_x, \text{val}_n)\). We can model the program with the control locations \( l_2, l_3, l_4, l_5 \) and \( l_7 \), each corresponding to the control location just before the line specified by the subindex is executed. Additionally, we have the control locations \( \text{end} \) for the end of the program, \( \text{err} \) where failed assertions lead, and \( \text{block} \) where failed assumptions lead. We then have

\[
\begin{align*}
\text{Loc} & = \{l_2, l_3, l_4, l_5, \text{end}, \text{err}, \text{block}\}, \\
\text{Val} & = \text{Int} \times \text{Int}, \\
\text{Loc}_{\text{init}} & = \{l_2\}, \\
\text{Val}_{\text{init}} & = \{(x, n) | x, n \in \text{Int}\}.
\end{align*}
\]
To understand the state transitions, we consider control location \( l_3 \). The transitions from the control location \( l_3 \) to \( l_4 \) are the mappings \((x, y) \mapsto (0, y)\) for all integers \( x, y \). Thus, \( \rightarrow \) contains the tuples
\[
\{(l_3, (x, y)), (l_4, (0, y))\} \mid x, y \in \text{Int}.
\]

Moreover, let \( \rightarrow_i \) denote the set of state transitions from control location \( l_i \), then
\[
\rightarrow = \bigcup_i \rightarrow_i.
\]

We note that there are no state transitions from the locations end, block and err. Figure 1 shows the program graph corresponding to this model.

Building on the intuition from program graphs, we can construct a set of CHCs representing the program. The idea is to construct the set so that their conjunction is satisfiable if and only if the error state is unreachable. Example 2.10 shows how this can be done.

**Example 2.10.** We will construct a set of CHCs based on the program graph from Example 2.9. Let each control location \( l_i \) denote an (uninterpreted) predicate. The initial state of the program can be any element in \( \text{Val}_{\text{init}} \) executing from any control location in \( \text{Loc}_{\text{init}} \), which we state in the initial clause:
\[
\text{true} \xrightarrow{l_2} (x, n).
\]

From \( l_2 \), an assumption is made, which can lead to two different behaviours. Either the assumption holds, in which case \( n \geq 0 \land l_2(x, n) \rightarrow l_3(x, n) \), or it does not, which indefinitely blocks the execution. Since blocking means the execution will never reach an error state, we need not consider this scenario any further. From \( l_3 \), an assignment is made, so \( l_3(x, n) \rightarrow l_4(0, n) \). From \( l_4 \), the while statement causes a branching, so we again model two clauses with constraints: \( x < n \land l_4(x, n) \rightarrow l_5(x, n) \), and otherwise \( x \geq n \land l_4(x, n) \rightarrow l_7(x, n) \). The clause from \( l_5 \) is modelled similarly to \( l_3 \). Finally, from \( l_7 \) an incorrect assertion leads to an error, so we model \( x \neq n \land l_7(x, n) \rightarrow \text{false} \). This ensures that if we get a satisfiability result, there is no execution such that \( x \neq n \) at \( l_7 \). If the assertion holds, we enter the end state which never leads to an error state, so this scenario does not need to be considered any further. Putting this together, we get the following set of CHCs:
\[
\begin{align*}
\text{true} & \rightarrow l_2(x, n) \\
n \geq 0 & \land l_2(x, n) \rightarrow l_3(x, n) \\
l_3(x, n) & \rightarrow l_4(0, n) \\
x < n & \land l_4(x, n) \rightarrow l_5(x, n) \\
x \geq n & \land l_4(x, n) \rightarrow l_7(x, n) \\
l_5(x, n) & \rightarrow l_4(x + 1, n) \\
x \neq n & \land l_7(x, n) \rightarrow \text{false}
\end{align*}
\]

Checking the satisfiability of these clauses in a Horn solver yields the result that the CHCs are satisfiable, so the program assertion holds. \( \square \)
2.2.4 The SMT-LIB Standard

The SMT-LIB Standard defines a common language and interface for SMT solvers. It allows us to use different background theories and common constructs to describe SMT problems. We will briefly describe a subset of the syntax for a few constructs that are sufficient to formulate simple sets of CHCs. Here, \(<\text{sort}>\) denotes a type such as integers, \(<\text{symbol}>\) denotes a symbol name, \(<\text{term}>\) denotes a variable or a function applied to terms, and \(<\text{logic}>\) denotes a collection of background theories.

- \((\text{set-logic } <\text{logic}>\)) defines what background theory applies in the clauses.
- \((\text{declare-fun } <\text{symbol}> (<\text{sort}>>) <\text{sort}>\)) declares a symbol corresponding to a function with the arguments given in parenthesis and the result given last.
- \((\text{assert } <\text{term}>\)) asserts that the specified term holds.
- \((\text{forall } ((<\text{symbol}> <\text{sort}>>) ...) <\text{term}>\)) evaluates a universal quantification. Existential quantification is expressed using \((\text{exists})\) and has a corresponding syntax.
- \((= <\text{symbol}> <\text{symbol}>\)) determines whether two elements are equal. Common first-order logic constructs like logical and \((\text{and})\), implications \((\Rightarrow)\) etc, have a corresponding syntax.

Example 2.11 shows how we can express the Horn clauses in Example 2.10 using the SMT-LIB standard.

Example 2.11. The CHCs in Equation \(4\) can be formulated in SMT-LIB as follows:

```plaintext
(set-logic HORN)
(declare-fun l2 (Int Int) Bool)
(declare-fun l3 (Int Int) Bool)
(declare-fun l4 (Int Int) Bool)
(declare-fun l5 (Int Int) Bool)
(declare-fun l7 (Int Int) Bool)

(assert (forall ((x Int) (n Int)) (=> true (l2 x n))))
(assert (forall ((x Int) (n Int)) (=> (and (>= n 0) (l2 x n)) (l3 x n))))
(assert (forall ((x Int) (n Int)) (=> (l3 x n) (l4 0 n))))
(assert (forall ((x Int) (n Int)) (=> (and (< x n) (l4 x n)) (l5 x n))))
(assert (forall ((x Int) (n Int)) (=> (and (>= x n) (l4 x n)) (l7 x n))))
(assert (forall ((x Int) (n Int)) (=> (l5 x n) (l4 (+ x 1) n))))
(assert (forall ((x Int) (n Int)) (=> (and (not (= x n)) (l7 x n)) false)))
(check-sat)
```
We note that using CHCs as an intermediate step allows for separating the model verification problem into two parts; encoding programs into CHC formulas and checking the satisfiability of CHC formulas.

### 2.2.5 Theory of Heaps

The theory of heaps (TOH) presented in [14] describes an SMT-LIB background theory for representing programs with heap and pointers, which includes performing operations such as allocating space, reading and writing data. This theory standardises a format for programs with heap data structures and enables the problem of heap reasoning to be passed from being part of the program modelling to being handled by Horn solvers.

To understand this theory, we need to look closer at the heap declaration and the functions of the theory. The heap declaration indicates the names of the sorts in the heap and what objects are allowed to be stored on the heap etc. In the declaration, algebraic data types (ADTs) are used. These are compositions of data types that allow for defining constructs such as records, unions and recursion-based data structures such as lists. Specifically, a basic heap declaration takes the following form:

\[
(\text{declare-heap } c_h \ c_a \ c_o \ \tau_o \ ((\delta_1 \ k_1) \ ... \ (\delta_n \ k_n)) \ (d_1 \ ... \ d_n))
\]

where \(c_h\) is the name of the declared heap sort, \(c_a\) is the name of the heap address sort, \(c_o\) is the name of the object sort allowed on the heap, \(\tau_o\) is the name of the default object, \(\delta_i\) is the name of algebraic data type \(i\), \(k_i\) is the arity of \(\delta_i\), and \(d_i\) is a list of constructors for \(\delta_i\) (if \(k_i > 0\), then the list \(l\) of constructors is wrapped with \((\text{par } (u_1 \ ... \ u_n) \ l)\), where the \(u_i\) are sort parameters) [14, 9].

The default object is a special object assumed to be stored at unallocated heap locations. This object is thus returned whenever a read from an unallocated address is attempted. A constructor has the syntax

\[
(c \ (s_1 \ \tau_1) \ ... \ (s_m \ \tau_m))
\]

where \(c\) is the name of the constructor, \(s_i\) is a selector and \(\tau_i\) is the sort corresponding to that selector. Example 2.12 illustrates a heap declaration in SMT-LIB.

**Example 2.12.** We will consider the heap declaration of a program from [13] which stores linked lists on the heap. The following statement declares a heap sort named Heap, with an address sort Addr which allows for storage of the sort Object. Object has three constructors including 0_Const, 0_Nil and 0_Empty, out of which the last one is the default object. 0_Const represents a node of a linked list and 0_Nil represents the end of such a list. These objects are built on the algebraic data types IntList, Cons and Nil which are also defined in the statement.

\[
\text{(declare-heap } c_h \ c_a \ c_o \ \tau_o \ ((\delta_1 \ k_1) \ ... \ (\delta_n \ k_n)) \ (d_1 \ ... \ d_n))
\]

where \(c_h\) is the name of the declared heap sort, \(c_a\) is the name of the heap address sort, \(c_o\) is the name of the object sort allowed on the heap, \(\tau_o\) is the name of the default object, \(\delta_i\) is the name of algebraic data type \(i\), \(k_i\) is the arity of \(\delta_i\), and \(d_i\) is a list of constructors for \(\delta_i\) (if \(k_i > 0\), then the list \(l\) of constructors is wrapped with \((\text{par } (u_1 \ ... \ u_n) \ l)\), where the \(u_i\) are sort parameters) [14, 9].

The default object is a special object assumed to be stored at unallocated heap locations. This object is thus returned whenever a read from an unallocated address is attempted. A constructor has the syntax

\[
(c \ (s_1 \ \tau_1) \ ... \ (s_m \ \tau_m))
\]

where \(c\) is the name of the constructor, \(s_i\) is a selector and \(\tau_i\) is the sort corresponding to that selector. Example 2.12 illustrates a heap declaration in SMT-LIB.
A heap declaration implicitly defines one additional sort, the \textit{ARHeap} consisting of a pair of \textit{(Heap, Address}). A result of this sort is returned when an allocation to the heap is made. To perform heap-related operations, the following functions are used:

- \textbf{nullAddress} : () \rightarrow \textit{Address}, returns an address that is always unallocated.

- \textbf{emptyHeap} : () \rightarrow \textit{Heap}, returns the heap that is unallocated everywhere.

- \textbf{allocate} : \textit{Heap} \times \textit{Object} \rightarrow \textit{Heap} \times \textit{Address}, returns a pair on the form \langle \textit{Heap, Address} \rangle where the returned heap has stored the given object at the returned address. Previously allocated locations remain unchanged.

- \textbf{valid} : \textit{Heap} \times \textit{Address} \rightarrow \textit{Bool}, returns true if and only if the location at the input address was previously allocated in the heap.

- \textbf{read} : \textit{Heap} \times \textit{Address} \rightarrow \textit{Object}, returns the object at the given address and heap if the address is valid, otherwise the default object is returned.

- \textbf{write} : \textit{Heap} \times \textit{Address} \times \textit{Object} \rightarrow \textit{Heap}, returns a heap with the given object at the given address if the address is valid, otherwise the heap is returned unchanged.

An example heap expression is given in Example 2.13.

\textbf{Example 2.13.} Define selectors \texttt{newHeap} and \texttt{newAddr} for the \textit{ARHeap} sort, and let the \textit{Object} sort have a constructor defined by \texttt{(O \_Int \_getInt \_Int)}. A heap state can then be expressed as a sequence of \texttt{allocate} and \texttt{write} operations applied to an \texttt{emptyHeap}.

```plaintext
1 write(newHeap(allocate(emptyHeap, 0 \_Int(1))),
2       newAddr(allocate(emptyHeap, 0 \_Int(1))),
3       0 \_Int(2))
```
This expression represents a heap where the first address contains the *Object* $O_{\text{Int}(2)}$ which overwrote the previous value $O_{\text{Int}(1)}$.

### 2.3 Contract Inference

As stated in Section 2.1.7, the deductive verification process consists of formalising specifications for a code base in the form of a main contract and then providing function contracts and other annotations that are sufficient for a deductive verifier to prove the correctness of the code. In [3], a method is proposed for utilising model checking techniques to automatically infer contracts. As in Example 2.2, we can use Hoare rules to model a program verification problem as a satisfiability problem on a set of clauses. By including an additional Hoare rule for procedure calls, and expressing the set of clauses as CHCs, a Horn clause solver can be instructed to find a solution involving predicates representing preconditions and postconditions. Such solutions can be translated to ACSL contracts and used for deductive verification.

#### 2.3.1 TriCera

TriCera [12] is a tool implementing model checking methodology similar to what was described in Section 2.2.3 which can be used to infer contracts for programs written in a subset of the C11 standard. Specifically, TriCera transforms a given program into CHCs, which are sent to the Horn solver Eldarica, returning a safety guarantee or a counter-example showing how an error state can be reached. Additionally, if the program is safe, then computed preconditions and postconditions for procedure calls can be translated into ACSL contracts. Initially, the computed pre- and poststates are given as syntax trees using the TOH described in Section 2.2.5 to handle pointers and heap data. In these expressions, the heap object constructors play an important role as a bridge to the TOH domain. For each sort $\text{Sort}$ that lives on the heap, TriCera defines the constructor $(O_{\text{<Sort>}} (\text{get<Sort> Sort}))$. Thus, for sorts declared to live on the heap, we note the following:

- $O_{\text{<Sort>}}$ constructs a heap *Object* containing a $\text{Sort}$ value. This is called a wrapper.

- $\text{get<Sort>}$ is a selector to get the value of a heap *Object* wrapping a $\text{Sort}$ value. This is called a getter.

Another important feature of contract formulas computed by TriCera is how the pre- and post-state heap are expressed. This is done using the notation @h to refer to the current heap and \old(@h) to refer to the old heap (if it exists). TriCera also defines the selectors newHeap and newAddr for getting the corresponding values from ARHeap objects returned by the TOH allocate function. To get the address of the last allocated address of a TOH heap, the selector counterAddr can be used. Additionally, to check whether some heap object obj has sort number n (by the order of definition in the heap declaration),
the selector HeapObject ctor is used on the form HeapObject ctor(obj) = n. For simplicity, we will express such statements on the form TriCERA prints it for readability, namely is_0<Sort>(obj).

At its current state, TriCERA does not support translating contracts with TOH statements to ACSL contracts. This thesis aims to provide this support.

### 2.3.2 Saida

Saida is a Frama-C plugin which facilitates usage of the contract inference method described in Section 2.3. One of the main problems it solves is that the model checking strategy described in Section 2.2 uses assertions and assumptions for expressing code specifications, whereas the deductive verification procedure from Section 2.1.7 relies on contracts expressed in ACSL. In [4], a translation is described from ACSL contracts to a subset of the C language extended with assert and assume statements. This translation is implemented in Saida [5] which allows for automatic translation of the specifications in a user-provided main contract to a harness function which TriCera can interpret. The Saida plugin can then send the code to the TriCera model checker, and automatically insert the inferred ACSL contracts into the code. Example 2.14 shows an example usage of the Saida plugin.

**Example 2.14.** Consider the code in Listing 6 which presents a C version of the set x procedure accompanied by a main procedure and the corresponding main contract. We will use the Saida plugin to infer a contract for the set x procedure. When running the plugin on this code, the program is first rewritten so that the contract is expressed in a harness function with assume and assert statements as is presented in the main procedure of Listing 7. Note that the procedure main of Listing 6 is renamed to main2 in Listing 7 so that the harness function can be the entry point for TriCera. Then, a contract for the set x procedure can be inferred by TriCera which is presented in Listing 8.

### 2.4 Related Work

This section discusses other work in contract inference for deductive verification and techniques for reasoning about heap constructs. We consider three approaches to static inference, which consists in performing analysis directly on code, and then dynamic inference, where also results of program execution are considered.

#### 2.4.1 Automatic Contract Inference

The first static approach to contract inference is property-guided inference, which consists of methods that attempt to prove properties defined by some specification. The method for contract inference considered in this thesis falls into this category; we attempt to show the properties defined by the main contract.
Listing 6 Procedure main with ACSL contract

```c
1 int x;
2 int n;
3 void set_x() {
4     x = 0;
5     while (x < n) {
6         x = x + 1;
7     }
8 }
9
10/*@ requires n >= 0;
11  ensures x >= 0;
12  assigns x;*/
13
14 void main() {
15     set_x();
16 }
17```

Listing 7 Harness function generated by SAIDA

```c
1 int x;
2 int n;
3 /*@contract@*/
4 void set_x() {
5     x = 0;
6     while (x < n) {
7         x = x + 1;
8     }
9 }
10
11 void main2() {
12     set_x();
13 }
14
15 extern int non_det_int();
16
17 void main() {
18     x = non_det_int();
19     n = non_det_int();
20     assume(n >= 0);
21     main2();
22     assert(x >= 0);
23 }
24```

Listing 8 Contract inferred for set_x by TriCera

```c
1 /*@*
2 requires n >= 0;
3  ensures n == \old(n) && x >= \old(n) && \old(n) >= 0;*/
The strength of such methods is that they do not necessarily have to cover complicated corner cases but rather infer contracts that are just sufficient to show some property. On the flip side, the reusability of these contracts may be limited. Another method that fits into this category is maximal specification synthesis [1], which given a postcondition Q and a program S that calls the procedures $f_1, ..., f_n$, attempts to find the most permissive specifications for $f_1, ..., f_n$ that ensures the correctness of S. This is done by using a counterexample-guided inductive synthesis loop to reduce the problem to a problem the authors call multi-abduction, which can be solved with a proposed algorithm.

The second approach of consists of methods aiming to compute the weakest precondition as described in 2.1.3. This idea was originally presented in [10], together with the weakest precondition calculus to compute a weakest precondition given a program and a postcondition. A fundamental issue with this approach is that for programs with loops, the weakest precondition is not always computable [9]. Another approach to computing weakest preconditions is by using abstract interpretations, a technique that builds on over-approximating program properties. In [29], a method for precondition inference is presented which is based on counter-example guided abstraction refinement (CEGAR). A similar approach is presented in [30] which outperforms the previous method. Neither of the implementations in [29] and [30] support reasoning about heap properties which is stated to be out of the scope for the tool due to the lack of a theory for reasoning about heaps.

Third, and complementary to weakest preconditions are strongest postconditions. Techniques based on this concept conversely attempt to compute the strongest postcondition given a program and optionally a precondition. Should no precondition be available this can be set to true so that the postcondition for any execution will be computed. In [15], such a technique is presented based on symbolic execution. However, this technique struggles with loops, which may lead to arbitrarily long execution paths. Problems like these are expected as the strongest precondition, just like the weakest precondition, is not always computable. In [2], a symbolic execution-based method is used to synthesise contracts for programs with pointers and heap manipulation.

Dynamic techniques for contract inference consider the program state during execution at different program locations to draw conclusions about certain properties of the program. One tool that uses such a method is Daikon [11] which can be applied to different languages including for example C, Java and Eiffel. In [24], a study is conducted comparing inferred contracts with human-written contracts. The study found that Daikon inferred about 5 times more assertions than humans, it managed to infer 60% of the human-written contracts, and about one-third of the inferred contracts were incorrect or irrelevant. This technique was combined with symbolic execution in [32] to improve the quality of the inferred contracts.
2.4.2 Reasoning with Pointers and Heaps

Heap reasoning is a challenging problem for program verification and existing model checkers use various techniques to model pointers and heap constructs. For example, SeaHorn [16] uses a collection of arrays to represent the heap, whereas JayHorn [27] uses invariants. This stands in contrast to TriCera, the model checker used to infer contracts in this thesis, which uses TOH which is described in Section 2.2.5 and leaves the heap reasoning to the Horn solver.

Another related research area is separation logic [25], which is an extension of Hoare logic. This theory provides concepts for allocating, reading, modifying and deallocating shared storage, allowing for formal reasoning about pointers and heap constructs. A successful tool building on separation logic is Infer [8], which automatically verifies the memory safety of C programs, and is in fact included in the Facebook code inspection chain [18]. This tool is focused on proving a restricted set of properties and so does not require specifications to be provided.

3 Translating Theory of Heap Contracts

This section describes the methodology developed for translating TOH contracts generated by TriCera to ACSL. We start by describing a fundamental difference between TriCera and ACSL contracts, followed by a detailed description of the translation approach. After that follows a discussion regarding the implementation of the translation approach.

3.1 Implicit vs Explicit Heap Expressions

An important difference between expressions with TOH compared to ACSL is that the former describes any statements about the heap explicitly, whereas ACSL expresses properties about the heap implicitly. We recall that TriCera uses the notation $\@h$ and $\old(\@h)$ to refer to the current and old heap (if it exists) respectively. As an example of the difference between implicit and explicit heap expressions, let $a$ be a valid heap pointer and consider the two postconditions in Listings 9 and 10. These expressions contain essentially the same information. We see that in ACSL, any references to the heap are implicitly interpreted to refer to the current heap, whereas the old heap state (if it exists) is accessed using the $\old$ construct. In TOH, any heap-related expression explicitly refers to the pre-state or post-state heap.

<table>
<thead>
<tr>
<th>Listing 9</th>
<th>ACSL: Implicit Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>*a == 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 10</th>
<th>TOH: Explicit Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$@h == \text{write}(\old(@h), a, 0_\text{Int}(1))$</td>
<td></td>
</tr>
</tbody>
</table>
3.2 Translation Development

Due to the complexity of expressions involving TOH and the differences compared to ACSL, the development of a translation method was based on finding good translation heuristics. In fact, some properties are not expressible in ACSL (such as explicit address properties as will be discussed in Section 3.3.8), and thus the translation can not guarantee that all information is kept in the translation. By manually reading and understanding the contracts with TOH expressions, ideas for extracting sufficient information could be designed. By iteratively testing and improving ideas, the translation strategy presented in the following section was developed.

3.3 The Translation Strategy

This section provides the details of the strategy for translating TriCera expressions involving TOH to ACSL contracts.
3.3.1 Overview

The translation process for a contract is implemented as a pipeline of transformations, each of which is responsible for a specific task. Each step is applied to both the precondition and the postcondition before moving on to the next one. The pipeline is illustrated in Figure 2. Below follows a brief description of each step in the pipeline. The following sections will provide more details about each step.

- **PostconditionSimplifier**: In this step, attempts are made to simplify the postcondition by using the information in the precondition. This is done as the simplified postcondition may contain more clauses that are directly expressible in ACSL. The precondition is left unchanged.

- **PointerPropExtractor**: In this step, any separation and validity clauses that can be deduced are extracted. This can only be done whenever the heap state is expressed.

- **AssignmentExtractor**: In this step, postcondition equalities between dereferenced pointers and their values are extracted. This information is extracted from assignments to the heap and can only be done whenever the post heap state is expressed. The precondition is left unchanged.

- **&**: This step creates a conjunction of the precondition or simplified postcondition with the corresponding extracted clauses from the PointerPropExtractor and AssignmentExtractor.

- **TOHProcessor**: In this step, TOH expressions are reduced to a simpler form containing fewer subexpressions, which can be handled by the ADTProcessor and ACSLProcessor.

- **ADTProcessor**: In this step, struct expressions are reduced to a simpler form containing fewer subexpressions, which can be handled by the ACSLProcessor.

- **ToVarForm**: In this step, terms are translated to their corresponding program variable form. This is done to make more information expressible in ACSL.

- **ACSLProcessor**: In this step, TOH expressions are translated to ACSL expressions.

- **ClauseRemover**: In this step, clauses that are not expressible in ACSL are removed.

- **ACSLLineariser**: In this step, a contract is generated in the form of a text string from the processed syntax tree.
3.3.2 PostconditionSimplifier

As certain statements are not expressible in ACSL, it is sometimes helpful to consider the information of the precondition and postcondition together and try to make simplifications. A formula is considered a simplification of another formula if it has fewer subexpressions and is semantically equivalent to the other formula. In this translation step, an attempt is made to use the information of the precondition to simplify clauses in the postcondition. Whenever inexpressible parts are simplified away from a clause, that clause will become expressible in ACSL. To see an instance where this is useful, consider Example 3.1.

Example 3.1. Consider the contract in Listing 11. The postcondition is not directly expressible in ACSL, since it explicitly states the address value of a pointer. However, the precondition in the contract already states that the inexpressible property holds, which we can use to simplify the postcondition in order to make it expressible, as can be seen in Listing 12. The contract can then be translated as shown in Listing 13.

Listing 11 Contract with inexpressible postcondition

1. requires (counterAddr(@h) == b)
2. ensures (!(counterAddr(\old(@h)) == \old(b)) || (a >= \old(a) + getInt(read(@h, counterAddr(\old(@h)))))

Listing 12 Contract with expressible postcondition

1. requires (counterAddr(@h) == b)
2. ensures (a >= \old(a) + getInt(read(@h, counterAddr(\old(@h)))))

Listing 13 Translated contract

1. requires \true
2. ensures (a >= \old(a) + \*b)

Figure 3: Listings 11 and 12 show how the postcondition of a contract can be made expressible in ACSL using simplification. Note that the two contracts are equivalent. Listing 13 shows the proposed translation of the contract.

The strategy to make such simplifications consists in coming up with a simplified postcondition and then checking whether substituting this postcondition with the previous one yields an equivalent contract. More formally, denote the
precondition by $P$, the postcondition by $Q$ and the simplified postcondition by $Q'$. Then the contract is equivalent if and only if the formula

$$P \implies (Q \iff Q')$$

is valid (i.e., it holds for all assignments of free variables). This is checked using a theorem prover. If the contract is equivalent, then the simplified postcondition replaces the previous postcondition. Otherwise, the previous postcondition is kept. To come up with candidate simplifications, one boolean formula is replaced at a time by the true or false literal. The procedure is repeated for all possible replacements. Finally, any boolean literals are simplified away by using trivial simplification rules (e.g. $F \land true \iff F$).

A similar approach could be applied to the precondition as well. In that case, $P$ would be simplified to some formula $P'$ and the contract would be equivalent if and only if $P \iff P'$. This was not implemented in this work as a contract where this would be fruitful was never encountered.

### 3.3.3 PointerPropExtractor

In this step, an attempt is made to extract valid and separated clauses whenever the heap state is expressed. The idea is to express the current state of the heap as explicitly as possible, and then simulate each operation made to the starting heap in order to obtain a model of the current heap. This model is then used to extract relevant properties.

If the heap state is expressed in a contract, it is expressed on the form

$$@h == \text{heap}$$

where $@h$ is the current heap state and $\text{heap}$ is a TOH $\text{Heap}$. In TOH, there are three operations that return a $\text{Heap}$: $\text{emptyHeap}$, $\text{allocate}$ (via the $\text{ARHeap}$ sort) and $\text{write}$. Thus, any $\text{Heap}$ expression must be on the form

1. $\text{emptyHeap}$
2. $\text{newHeap}(\text{allocate}(\text{Heap}, \text{Object}))$
3. $\text{write}(\text{Heap}, \text{Address}, \text{Object})$

We can therefore conclude that any expression of the heap state in a contract can be expressed as a chain of $\text{write}$ and $\text{newHeap}$-allocate operations starting from an origin heap, denoted by $\text{originHeap}$, which consists of either the $\text{emptyHeap}$ or a quantified heap variable. However, variables are occasionally introduced to represent expressions like addresses, objects or heaps. Thus, any variables in the heap expression must be substituted by their explicit representations in order to obtain an explicit heap expression. Once this is done, we can simulate the operations applied to the $\text{originHeap}$ as defined by TOH. This is done by keeping a map from addresses to objects, and a counter of the number of allocated objects. Any allocated addresses are determined by the current value of the counter. When $\text{originHeap}$ is quantified, we do not know how much memory is allocated, so the counter keeps track of the memory allocated in addition to what was already allocated on the $\text{originHeap}$. Thus

---

32
addresses must keep track of which originHeap they refer to. If at some point a write occurs to an address that is not known to be allocated, all variables are havoced, i.e., removed from the map. The result of this simulation is a map from addresses to objects where the addresses are known to be allocated and point to a valid value. Additionally, the addresses are known to be separated. Thus, these addresses can be replaced by any corresponding pointer variables to construct and output \texttt{valid} and \texttt{separated} clauses.

Example 3.2. To give an understanding of how information is extracted, we consider the heap expression:

\begin{verbatim}
write(write(newHeap(allocate(emptyHeap, O_Int(1)),
newAddr(allocate(emptyHeap, O_Int(1))),
0_Int(2)))
nthAddr(2),
0_Int(3)) = \$h
\end{verbatim}

Let us iteratively replace heap objects with a heap model $HM(counter, Map())$ which keeps track of the number of allocated objects and what values different addresses currently map to. We start by replacing \texttt{emptyHeap} with a heap model:

\begin{verbatim}
write(write(newHeap(allocate(HM(0, Map()), O_Int(1),
newAddr(allocate(HM(0, Map()), O_Int(1))),
0_Int(2)))
nthAddr(2),
0_Int(3)) = \$h
\end{verbatim}

An object is allocated to the heap:

\begin{verbatim}
write(write(HM(1, Map(Addr(0) -> O_Int(1))),
Addr(0),
0_Int(2)))
nthAddr(2),
0_Int(3)) = \$h
\end{verbatim}

A write is made to the allocated address:

\begin{verbatim}
write(HM(1, Map(Addr(0) -> O_Int(2)))
nthAddr(2),
0_Int(3)) = \$h
\end{verbatim}

A write is made to an unallocated address. The heap is havoced:

\begin{verbatim}
HM(1, Map())
\$h
\end{verbatim}

In this case, we end up with an empty map, so no information could be extracted. Should key-value pairs remain in Map, we would know those addresses to be separated and valid.
Note that the latest version of TOH also supports deallocation, which is not considered in this thesis.

### 3.3.4 AssignmentExtractor

This step is only applied to the postcondition, and the aim of it is to extract equalities from the current heap state. As the WP plugin does not currently support ACSL clauses for specifying allocation and deallocation, procedures and contracts that use these operations are not considered in this thesis. Thus, with the argument from the previous section, the post-state Heap object is expressed in the form of a chain of write statements ending with the pre-state heap. This form makes it easy to reason about assignments made to the heap.

Consider any expression of the form

$$\text{addr}_n \leftarrow \text{obj}_n == \text{write}(...\text{write}(@h, \text{addr}_1, \text{obj}_1), ..., \text{addr}_n, \text{obj}_n).$$

This can be thought of as a sequence of assignments, where \(\text{obj}_1\) is assigned to the address \(\text{addr}_1\), then \(\text{obj}_2\) is assigned to the address \(\text{addr}_2\), and so on. This idea of a sequence of assignments can be used to extract equalities for the postcondition.

The procedure for extracting equalities is as follows. Consider the backwards sequence of assignments, \((\text{addr}_n \leftarrow \text{obj}_n), ..., (\text{addr}_1 \leftarrow \text{obj}_1)\). Form a subsequence \(\text{safeSeq}\) by taking assignments from this sequence while all taken addresses are valid and separated from each other. This can be done by using the separateness and validity information extracted by the PointerPropExtractor. Now, form a subsequence of \(\text{safeSeq}\) by taking only the assignments where the address is its first occurrence. The result will be the most recent assignments such that all addresses are unique, valid and separated. Thus, we know that the equalities from these assignments hold in the post-state. Now, for the addresses that have a corresponding pointer variable \(p\), we can extract an equality clause of the form \(\text{get<Sort>(read}(@h, p)) == \text{get<Sort>(obj)}\). Note that we express the equality of the actual values of the objects, not the heap objects. This is done since this is the form can be read by the ACSLProcessor.

**Example 3.3.** Suppose we know that \(\text{old}(a), b\) are separated and valid, and that \(c\) is an address we do not have any separateness or validity information about. Consider the following TRICERA formula:

```plaintext
(write(write(write(@h, \text{old}(a)), \text{old}(a), 0.Int(1)), c, 0.Int(42)), b, 0.Int(getInt(read(@h, \text{old}(a)))) = @h)
```

Reading this write-chain, we form a reversed sequence of assignments:
(b <- O_Int(getInt(read(@h, \old(a))))),
(c <- O_Int(42)),
(\old(a) <- O\Int(1))

Now, taking assignments while they are separated and valid yields the single-item sequence:

(b, O_Int(getInt(read(@h, \old(a)))))

Taking the first occurring write to each address is trivial here. We finally form the clause:

getInt(read(@h, b))
==
getInt(O_Int(getInt(read(@h, \old(a))))))

In the coming sections we will see how this expression can be translated to ACSL.

### 3.3.5 TOHProcessor

To reduce TOH expressions to a form that can be handled by the ADTProcessor and ACSLProcessor, simple translation rules are applied to the TOH expressions until a fixed point is reached. The rules are as follows: for any \texttt{O<Sort>} value \texttt{obj}, any \texttt{<Sort>} value \texttt{s}, any \texttt{Address addr} and any \texttt{Heap h}:

- \texttt{O<Sort>(getInt<Sort>(obj))} is replaced by \texttt{obj}
- \texttt{getInt<Sort>(O<Sort>(s))} is replaced by \texttt{s}
- \texttt{read(write(h,addr,obj),addr)} is replaced by \texttt{obj}

where \texttt{<Sort>} is any sort defined on the heap.

**Example 3.4.** Consider the resulting expression of Example 3.3:

\begin{verbatim}
getInt(read(@h, b))
==
getInt(O_Int(getInt(read(@h, \old(a))))))
\end{verbatim}

By applying the TOHProcessor we get:

\begin{verbatim}
getInt(read(@h, b))
==
getInt(read(@h, \old(a)))
\end{verbatim}

### 3.3.6 ADTProcessor

When structs are being processed in TOH, they frequently become nested in a way that makes it hard to extract information. An example of this is the following expression:

\begin{verbatim}
S(b(S(a(getS(read(@h, p)))), 42)), b(S(a(getS(read(@h, p)))), 42))
\end{verbatim}
which is equivalent to

\[ S(42, 42) \]

Again, a simple translation rule is applied until a fixed point is reached. For any struct \( S \) with selectors \( a_1, a_2, \ldots \) and any Objects \( obj_1, obj_2, \ldots \), the following rule is applied:

- \( a_i(S(obj_1, \ldots, obj_j, \ldots)) \) is replaced by \( obj_j \)

After this, for any expression \( s \), the following rule is applied to express each field equality independently:

- \( s == S(obj_1, obj_2, \ldots) \) is replaced by
  \[ a_1(s) == obj_1 \&\& a_2(s) == obj_2 \&\& \ldots \]

We remark the similarity between the first two rules of the TOHProcessor and the first rule of the ADTProcessor. Both apply similar simplification rules to ADTs. However, one important difference prevents merging the two into a single general rule, which is that the getters and wrappers considered in the TOHProcessor take only one argument. This makes wrapping the result of the getter an identity operation, which is not true for constructors taking multiple arguments.

### 3.3.7 ToVarForm

Since program variables can always be expressed in ACSL, it is desirable to translate all terms to their corresponding program variable form. This is done by reading equalities and replacing any terms possible. This is done as a preparation for the ACSLProcessor step.

**Example 3.5.** Consider the following precondition:

```plaintext
a == newAddr(allocate(emptyHeap, 0_Int(1)))
&& b == newAddr(allocate(newHeap(allocate(emptyHeap, 0_Int(1)), 0_Int(2)))))
```

And the following postcondition:

```plaintext
getInt(read(@h, newAddr(allocate(emptyHeap, 0_Int(1))))) ==
getInt(read(@old(@h), newAddr(allocate(allocate(emptyHeap, 0_Int(1)), 0_Int(2))))))
```
After applying ToVarForm, the postcondition is translated to the form

\[
\begin{align*}
\text{getInt} & \left( \text{read(@h, \old(b))} \right) \\
== \\
\text{getInt} & \left( \text{read(\old(@h), \old(a))} \right)
\end{align*}
\]

which can be handled by the ACSLProcessor.

### 3.3.8 ACSLProcessor

In this step, rules are applied to identify any dereference expressions that can be expressed in ACSL. We consider pointer dereferencing and struct field dereferencing. We start with expressions on the form \( \text{get<Sort>}(\text{read(heap, pointer)}) \), where \( \text{heap} \) is either the current (@h) or the old heap (\old(@h)), \( \text{pointer} \) is either a variable from the current state or the old state, and either a procedure argument \( a \) or a global variable \( g \). Depending on whether we are considering a precondition or a postcondition, we have the following translations:

- **Precondition:**

  \[
  \text{get<Sort>}(\text{read(}
  \begin{array}{cc}
  @h & a \\
  @h & g \\
  \end{array}
  \text{))} \rightarrow
  \begin{array}{c}
  *a \\
  *g \\
  \end{array}
  \]

- **Postcondition:**

  \[
  \text{get<Sort>}(\text{read(}
  \begin{array}{cc}
  @h & \old(g) \\
  @h & \old(a) \\
  \old(@h) & a \\
  \old(@h) & g \\
  \old(@h) & \old(a) \\
  \old(@h) & \old(g) \\
  \end{array}
  \text{))} \rightarrow
  \begin{array}{c}
  *\old(g) \\
  *a \\
  - \\
  - \\
  \old(*a) \\
  \old(*g) \\
  \end{array}
  \]

Continuing, for struct field dereferencing, we have expressions on the form \( \text{get<Sort>}(f(\text{read(heap, pointer)})) \), where \( f \) is a field selector and \( \text{heap} \) and \( \text{pointer} \) are on the same form as above. We have the following translations:

- **Precondition:**

  \[
  \text{get<Sort>}(f(\text{read(}
  \begin{array}{cc}
  @h & a \\
  @h & g \\
  \end{array}
  \text{))} \rightarrow
  \begin{array}{c}
  a\rightarrow f \\
  g\rightarrow f \\
  \end{array}
  \]
• Postcondition:

<table>
<thead>
<tr>
<th>Heap</th>
<th>Pointer</th>
<th>ACSL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>@h</td>
<td>a</td>
<td>-</td>
</tr>
<tr>
<td>@h</td>
<td>g</td>
<td>g-&gt;f</td>
</tr>
<tr>
<td>@h</td>
<td>@old(a)</td>
<td>a-&gt;f</td>
</tr>
<tr>
<td>@old(h)</td>
<td>a</td>
<td>\old(a)-&gt;f</td>
</tr>
<tr>
<td>@old(h)</td>
<td>g</td>
<td>\old(g)-&gt;f</td>
</tr>
<tr>
<td>@old(h)</td>
<td>@old(a)</td>
<td>\old(a-&gt;f)</td>
</tr>
<tr>
<td>@old(h)</td>
<td>@old(g)</td>
<td>\old(g-&gt;f)</td>
</tr>
</tbody>
</table>

These rules are applied to the pre- and postconditions in this step.

Example 3.6. Applying the ACSLProcessor to the result from Example 3.4, assuming $b$ is a global variable, and $a$ is a procedure argument, we have:

```
getInt(read(@h, b)) == getInt(read(@old(h), @old(a)))
```

which translates to

```
*b == \old(*a)
```

3.3.9 ClauseRemover

At this step of the pipeline, we want to remove any clauses that are not expressible in ACSL. First, any remaining clauses that contain TOH expressions are removed. Further, we consider clauses that explicitly state something about the address value of a pointer (e.g. for some pointer $p$, $p \geq 1$). Within TOH, such expressions may for example be used to reason about pointer aliasing and validity. However, since different compilers may use different rules for memory allocation, explicit address values do not in general translate to ACSL. A special case which is kept are relations between pointers on the form $p_1 == p_2$. Additionally, any trivial equalities on the form $a == a$ and heap expressions $@h == ...$ are removed.

Example 3.7. Consider the following postcondition:

```
ensures p >= 1 && p == q && d == getInt(read(..)) && *b == \old(*a)
```

By applying the ClauseRemover step, we get:

```
ensures p == q && *b == \old(*a)
```

3.3.10 ACSLLineariser

This step was already implemented in TRICERA and consists in translating the syntax tree to a string representation of the contract. Translations are added for nodes representing $\valid, \separated$, pointer dereferencing and struct field dereferencing.
3.4 Implementation

The implementation was done by extending TriCera with modules for the processing steps mentioned above. The modules were implemented in Scala and the version of the implementation used in this report can be found on Github [32].

An essential part of the implementation was a data structure for representing equalities between expressions. This was implemented as a set of Values, where each Value consisted of a set of expressions corresponding to the different forms that Value could be expressed in. This made it possible to read equalities from an expression and merge Values together when they were found to be equal. Further, methods for accessing various forms of a Value were implemented, such as the variable form, the address representation (introduced by the PointerPropExtractor) or a more explicit form (implemented by preferring neither variable nor address forms).

In the implementation of PostconditionSimplifier, one subformula at a time is replaced by true, and then the equivalence check described in 3.3.2 is performed using the Princess theorem prover [26] with a time limit of 100ms. The procedure is then repeated for replacements with false. Finally, trivial simplifications relating to the replaced true literals were made. Such trivial simplifications were never implemented for false literals since such a simplification was never encountered.

For the PointerPropExtractor, the current implementation for obtaining the most explicit form of the heap could give rise to infinite translation loops. The explicit form is obtained by first iterating over all equalities that must hold and forming a set of Values as described above. An explicitness ordering is then formed as:

1. Address representations (in the simulated heap)
2. Variables
3. Other expressions

where a low number corresponds to low explicitness and a high number to high explicitness. Using this ordering, a map is created so that each Value that has an explicit form adds mappings to one of the explicit forms from all of the other forms of that Value. This map is then applied to the formula until a fixed point is reached. It may be the case that this can give rise to cyclic mappings causing non-terminating translations. This has however not been encountered yet. However, if that is the case, a more careful ordering of explicitness could be a possible solution, where replacements are only made to higher levels of explicitness.

The ToVarForm module uses an expression replacement approach similar to the one used in PointerPropExtractor, but for replacing expressions with variables. In this case, no infinite loops can occur. This can be shown by arguing that any two variables mapping to each other (e.g., \(a \rightarrow b, b \rightarrow a\)) would belong to the same Value by the construction of the map. But since all
members of that Value are mapped to the same variable by the construction of the map, we get a contradiction. This argument can be extended to a mapping circle of any size.

To remove all occurrences of explicit pointer value expressions in the ClauseR-remover, all clauses containing pointers not wrapped inside a function application were removed, with the exception of the pointer equality case \( p1 == p2 \).

The implementations of AssignmentExtractor, TOHProcessor, ADTProsessor, ACSLProcessor and ACSLProcessor follow closely the descriptions from Section 3.3 and did not present any major difficulties.

4 Evaluation

The performance evaluation of the proposed translation strategy consisted in creating a test set of verification tasks, running Saida on each of these files, then performing some post-processing, and finally making a verification attempt with the translated contracts using the Frama-C WP plugin. This section gives the details of this procedure.

4.1 Verification Tasks

In order to evaluate the performance of the translation method, a set of verification problems was needed in the form of code samples with a main contract and procedures to infer contracts for. Additionally, the code samples needed to feature pointers and heap manipulation, and be in a form compatible with the Saida plugin. Since no set of code samples sufficiently close to meeting these requirements could be found, the evaluation set was constructed manually. To get an idea of the limits of the verification approach, some samples that could successfully be verified with the inferred contract were updated with a new more complicated version. The previous version was kept in the data set. To make use of non-determined values in allocated variables with the \(-lib-entry\) option of Frama-C, pointers to such values would have a corresponding global initialisation variable. Also obeying the form requirements of Saida, the verification samples had the following form:

```c
<function definitions>

<type>* a;
<type>* b;
...
<type> a_init;
<type> b_init;
...

/*@ 
requires ...
ensures ...
assigns a, a*, b, b*, ...
*/
```
int main(void) {
    a = malloc(sizeof(*a));
    b = malloc(sizeof(*b));
    ...
    *a = a_init;
    *b = b_init;
    ...
    <main code>
}

4.2 Contract Inference
To infer contracts for the code samples, the SAIDA plugin was run with its
-lib-entry option, and with the TOH translation version of TriCera set to
start with an arbitrary heap state. A time limit of 2 minutes was set for each
run. The program executions were run on a 2017 Macbook Pro with a 2,3 GHz
Dual-Core Intel Core i5 processor and 8 GB of RAM.

4.3 Post-Processing
To see if the contracts were sufficiently strong to prove the properties of the
main function, the following post-processing steps had to be performed:

1. Add #include<stdlib.h> to the top of the file to include the malloc
   function contract. This line could not be added initially as it causes a
   syntax error in SAIDA.

2. Add assigns clauses for all the inferred contracts to specify the frame
   condition.

3. Add admit clauses indicating that allocated pointers of the main function
   are valid and separated (if that is the case).

4. Move declarations of variables occurring in any inferred contract so that
   they are in context.

   We note that adding admit clauses for the contracts has to be done since
   malloc does not guarantee that a valid address is returned. Also, the problem
   of contract variables not being in context is a problem that needs to be ad-
   dressed, but is not rooted in the translation strategy. However, for the purpose
   of evaluating the translation strategy, this is not an issue.

4.4 Verification Attempt
After post-processing, the frama-c -wp command could be run on the resulting
file to see if the properties of the main function could be proven. The run
was deemed successful if all the properties except the main assigns clause were
proven. The assigns clause could generally not be proven due to the allocation
of new memory within the main function, which is acceptable for our purposes.
<table>
<thead>
<tr>
<th>Test File</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>get-1.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>get-2.c</td>
<td>Incorrect syntax</td>
</tr>
<tr>
<td>incdec-1.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>incdec-2.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>incdec-3.c</td>
<td>Insufficient translation</td>
</tr>
<tr>
<td>max-1.c</td>
<td>Insufficient translation</td>
</tr>
<tr>
<td>max-2.c</td>
<td>Insufficient translation</td>
</tr>
<tr>
<td>multadd-1.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>multadd-2.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>multadd-3.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>truck-1.c</td>
<td>Verifies</td>
</tr>
<tr>
<td>truck-2.c</td>
<td>TriCERA did not generate a contract within 2 minutes</td>
</tr>
</tbody>
</table>

Table 1: The results from performing the evaluation procedure on the test code samples.

5 Results

After constructing 12 code samples and performing the evaluation procedure on each, the results in Table 1 were obtained. Out of 12 test cases, 7 main contracts were successfully verified by following the evaluation procedure. Below follows comments on the unsuccessful test cases:

- **get-2.c**: The translated contract contains quantification expressions for variables of internal heap theory sorts, which is a syntax error. As the subexpression of the quantification expression did not contain any quantified variables, an attempt was made to remove the quantification, after which the program verified successfully.

- **incdec-3.c**: In this contract, important information was lost in the translation due to a subformula containing a read from an unknown heap (which was thus not considered). The contract stated that reading from the specific heap with the specific address would return an int, so more information could be extracted.

- **max-1.c**: The translated postcondition could not be proved. Two equalities relating pointer arguments to their values in the precondition were left out as they were expressed via the pre-state of the heap. After adding these equalities, the program was verified successfully.

- **max-2.c**: In this contract, important information was missed as the ToVarForm step could not find any variable corresponding to the expression counterAddr(0h).
• truck-2.c: In this test case, TriCera did not manage to compute a contract within the time limit, so the translation could not be applied.

The sample files and outputs from each translation step are available on Github [31].

5.1 Discussion

The results show that the translation strategy can extract sufficient information for simple code samples but that there are still weaknesses that need to be addressed. The successfully verified code samples featured recursion, aliasing and manipulation of heap-allocated structs.

One problem with the translation seems to have been related to quadratic expressions in the main contract. The only difference between get-1.c and get-2.c was that the postcondition was given a quadratic term, see Figure [14] and [15]. This suggests that TriCera introduces additional quantifiers to handle the quadratic expression. The issue that arises is that the additional quantifiers are over types only known within the TOH. This points out that a sort check needs to be done for quantifiers so that clauses containing such expressions can be removed in the ClauseRemover. In cases where the quantified variable does not occur in the subexpression of the quantifier, the subexpression could instead replace the quantifier in the syntax tree, thus keeping potentially valuable information.

Another problem occurred in the test case incdec-3.c, where the only new feature compared to incdec-2.c was a function call being made from within another non-main function. This feature probably caused expressions in the form of read operations from an unknown (quantified) heap, which the current translation strategy does not handle. Thus, important information was lost in the translation of this contract. However, the contract contained validity information about this read given on the form is_0.<Sort>(read(h,p)), where h is a TOH Heap and p is TOH Address. This problem could thus potentially be solved by extracting validity information on this form.

Further, the test case max-1.c revealed that important equalities in the precondition are sometimes expressed in the heap state. Thus, the AssignmentExtractor needs to be modified so that it can be applied to the precondition as well. However, as the pre-state heap expression can contain allocate operations, this requires handling a chain of both write and allocate operations. One potential way of doing this would be to model the heap through a simulation of the operations as in the PointerPropExtractor. This was looked at

<table>
<thead>
<tr>
<th>Listing 14 Postcondition of get-1.c</th>
<th>Listing 15 Postcondition of get-2.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ensures r1 &gt;= n_init &amp;&amp; r1 &lt;= n_init;</td>
<td>1. ensures r1 &gt;= n_init &amp;&amp; r1 &lt;= n_init * n_init;</td>
</tr>
</tbody>
</table>
in this project, but not carried through due to time constraints. The current strategy for simulating heap operations is to first express the heap expression in an as explicit form as possible, as mentioned in Section 3.3. However, a better approach could be to obtain the explicit form lazily, retrieving equal forms when needed, to be able to obtain more accurate contracts. For instance, suppose that $0_{\text{Int}}(1) \equiv \text{getInt(read}(\text{@h}, p))$, and that the explicit form contains the expression $\text{write}\left(\text{old}(\text{@h}), \text{nthAddr}(1), 0_{\text{Int}}(1)\right)$. Here, we don’t know whether the int literal or read statement was originally the write value due to how the explicit form translation works. However, this information could be kept by lazily obtaining the explicit form, allowing a more accurate translation of the assignment.

The only test case where it is difficult to propose a direct solution for the translation is max-2.c. In this case, important information was expressed in relation to $\text{counterAddr}(\text{@h})$ which can be interpreted as "the last allocated address of the current heap". By checking which was the last allocated variable in the program, it is easy to conclude which pointer this expression refers to (the program pointer variable $r$). However, no equality between the program variable and the $\text{counterAddr}(\text{@h})$ expression occurs in the contract. Thus, a solution would likely require either reading the program file or forcing TriCera to relate $\text{counterAddr}$ expressions to program variables whenever possible.

For the test case truck-2.c no contract could be inferred by TriCera. Thus, the translation strategy was not the limiting factor in this case.

Overall, the results confirm that the translation approach serves as a good starting point for translating expressions involving TOH, but that there are improvements to be made.

6 Conclusion

A strategy for translating formulas involving TOH has been proposed in Section 3.3. With a heuristic approach, the strategy consists of a number of steps, each extracting information or reshaping the expressions to finally output an ACSL contract.

Evaluating the implementation of the translation strategy on a collection of manually written verification tasks showed that the translation strategy can successfully translate simple code samples. However, the strategy still has weaknesses to address. In its current form, the translation strategy performs well on test programs featuring recursion, aliasing and manipulation of heap-allocated structs. On the other hand, it was unsuccessful in translating contracts with the following features:

- Quantification over TOH internal sorts.
- Reads from unknown heaps with validity deducible from expressions on the form $\text{is} _O , \text{<Sort>} (\text{read}(h,p))$.
- Important information contained in the heap state of the precondition.
• Important information being expressed in terms of the last allocated address (i.e., `counterAddr(@h)`) where it is unrelated to any program variable in the contract.

Improvements to overcome all of these weaknesses have been suggested, where all but the last case consist of direct improvements on the translation strategy. With this, we conclude that model checking tools can be used to infer contracts for deductive verification of programs with pointers and heap allocations, but currently to a limited extent. However, through improvements to the translation approach presented in this thesis, the extent to which contracts can be inferred will likely improve.

6.1 Limitations

The proposed translation strategy has a number of limitations:

• The current translation strategy does not support arrays and the related heap operations `batchAllocate` and `batchWrite` etc.

• The current translation strategy does not support the TOH `valid` function since it never occurred in a test case.

• The current translation strategy does not support allocations and deallocations within the function it is inferring a contract for. This constraint was set since the WP plugin does not currently support specifying allocations and deallocations within functions.

• The current translation strategy has been evaluated on a small, manually written data set of simple programs. Further testing might reveal unidentified shortcomings of the current approach.

An identified limitation in the contracts generated by TriCera is that cases occur where expressions are made in terms of the last allocated address, but this address is never related to any program variable. This seems to make certain expressions impossible to translate, where the only information missing is which program variable points to the last allocated address. Another more general limitation regarding inferred contracts from TriCera is that variable declarations occasionally need to be moved to be within the scope of the contracts.

6.2 Future Work

With the proposed translation strategy being a starting point for translating contracts involving TOH, there are improvements to be made. This section discusses such ideas.
6.2.1 Improved ClauseRemover

The ClauseRemover could be updated to remove any quantification expressions over sorts internal to TOH. If the quantified variable does not occur in the quantification subexpression, then the subexpression should be kept to preserve potentially valuable information.

6.2.2 Assigns Clauses

By taking inspiration from the heap state simulation of the PointerPropExtractor or the AssignmentExtractor, assigns clauses could possibly be computed. This would automate yet one more step of the verification process. However, since explicit heap expressions do not always occur in contracts computed by TriCera, this is not always possible.

6.2.3 Extracting More Validity and Separateness Information

As suggested by the verification problem incdec-3.c, expressions on the form isO<*Sort>(read(h,p)) imply that p is valid. Such information and explicit pointer separateness (e.g. p1 != p2) could be specified in the translated contract and also be used to get out more information from the PointerPropExtractor and AssignmentExtractor.

6.2.4 A More General Heap Extractor

As the issues with max-1.c indicate, the AssignmentExtractor could extract valuable information from the precondition by being able to handle allocate operations. One way of doing this could be to simulate the heap state as in the PointerPropExtractor and extract information about equalities from there. One suggested difference is to not translate the heap expression to its explicit form initially, but rather do this lazily. This way, more exact information about assignments can be kept. For instance, if a pointer dereference is equal to the integer 1, by keeping the initial form of the heap expression, it would be possible to deduce whether an assignment wrote the dereferenced pointer value or the integer value. This way, the final contract will more accurately capture what the function does. If the heap simulation approach could be used in the AssignmentExtractor, it would also add support for allocations in functions once the WP plugin starts supporting this feature. This approach also suggests that the PointerPropExtractor and AssignmentExtractor could be merged into a more general HeapExtractor. A further development would be to consider deallocations.

6.2.5 Further Testing

In order to be able to build a robust translation strategy, a larger and more diverse collection of verification tasks would be valuable. This could serve as a
benchmark to evaluate the performance of the translation strategy or be used to find weaknesses in the strategy and implementation.
References


A Appendix: Code Samples

A.1 get-1.c

Input file:

```c
int r1;
int n_init;

int get(int* p) {
  if (*p <= 0) {
    return 0;
  } else {
    *p = *p - 1;
    return 1 + get(p);
  }
}
```

```c
int* n;
/*@ requires n_init > 0;
ensures r1 >= n_init && r1 <= n_init;
assigns r1, n, *n;
*/
int main(void) {
  n = (int*) malloc(sizeof(*n));
  *n = n_init;
  r1 = get(n);
}
```

Inferred contract for `get`:

```c
/*@ requires p == n && n_init >= 1;
ensures ((result == 0 && 0 >= \old(*p)) || \result >= 0) && r1 == \old(r1) && n_init == \old(n_init) && \old(n) == n;
*/
```
A.2 get-2.c

Input file:

```c
int r1;
int n_init;

int get(int* p) {
  if (*p <= 0) {
    return 0;
  } else {
    *p = *p - 1;
    return 1 + get(p);
  }
}

int* n;
/*@requires n_init > 0;
ensures r1 >= n_init && r1 <= n_init * n_init;
assigns r1, n, *n;*/
int main(void) {
  n = (int*) malloc(sizeof(*n));
  *n = n_init;
  r1 = get(n);
}
```

Inferred contract for `get`:

```c
/*@requires p == n && (n_init >= 2 || (n_init >= 1 && *n == n_init
&& (n_init != 1 || \forall int v0; n_init == v0 - 1) && (n_init != 1 || \forall int v0; \exists HeapObject v1, v2; ((v2 == v1 || v2 == v1) && v2 == v1)) && (1 >= n_init || *p == n_init || *p >= 1)) || (n_init >= 1 && *p == n_init - 1 && (n_init != 1 || \forall int v0; n_init == v0 - 1) && (n_init != 1 || \forall int v0; n_init == v0 - 1)) || \forall int v0; n_init == v0 - 1) && (n_init != 1 || \forall int v0; n_init == v0 - 1) && (n_init != 1) || (1 >= n_init || *p == n_init || *p >= 1));
ensures ((result == 0 && 0 >= \old(*p)) || (result == 1 && \old (*p) == 1) || (((result == 5 && \old (*p) >= 3) || 2*\old(*p) - result >= 2) && \result > \old(*p))) && r1 == \old(r1) && n_init == \old(n_init) && \old(n) == n;*/
```
A.3 incdec-1.c

Input file:

```c
int a_init;
int b_init;

void increment(int* val) {
    (*val)++;
}

void decrement(int* val) {
    (*val)--;
}

int *a;
int *b;
/*@ requires \true;
ensures *a == a_init + 1 && *b == b_init - 1;
assigns *a, *b, a, b;
*/
int main(void) {
    a = (int*)malloc(sizeof(*a));
b = (int*)malloc(sizeof(*b));
*a = a_init;
*b = b_init;
increment(a);
decrement(b);
}

Inferred contract for increment:
/*@
requires val == a && \separated(val, b) && \valid(val) && \valid(b);
ensures \old(val) == a && a_init == \old(a_init) && b_init == \old(b_init) && \old(val) == \old(a) && b == \old(b) && \separated(val, b) && \valid(val) && \valid(b) && *val == 1 + \old(*val);
*/

Inferred contract for decrement:
/*@
requires val == b && \separated(a, val) && \valid(a) && \valid(val);
ensures \old(val) == b && a_init == \old(a_init) && b_init == \old(b_init) && a == \old(a) && \old(val) == \old(b) && *val == \old(*val) - 1;
*/
A.4 incdec-2.c

Input file:

```c
int a_init;

void increment(int* val) {
    (*val)++;
}

void decrement(int* val) {
    (*val)--;
}

int *a;
/*@ 
requires \true;
ensures *a == a_init;
assigns *a, a;
*/
int main(void) {
    a = (int*) malloc(sizeof(*a));
    *a = a_init;
    increment(a);
    decrement(a);
}
```

Inferred contract for `increment`:

```c
/*@ 
requires val == a && \valid(val);
ensures \old(val) == a && a_init == \old(a_init) && \valid(val) 
    && *val == 1 + \old(*val);
*/
```

Inferred contract for `decrement`:

```c
/*@ 
requires val == a;
ensures \old(val) == a && a_init == \old(a_init) && \old(val) == 
    \old(a) && *val == \old(*val) - 1;
*/
```
A.5 incdec-3.c

Input file:

```c
int a_init;
int b_init;
void increment(int* val) {
    (*val)++;
}
void decrement(int* val) {
    increment(val);
    *val = *val - 2;
}
int *a;
/*@ requires \true;
ensures *a == a_init;
assigns *a, a; */
int main(void) {
    a = (int*) malloc(sizeof(*a));
    *a = a_init;
    increment(a);
    decrement(a);
}
```

Inferred contract for `increment`:

```c
/*@ requires val == a;
ensures \old(val) == a && a_init == \old(a_init) && b_init == \old(b_init) && *val == 1 + \old(*val); */
```

Inferred contract for `decrement`:

```c
/*@ requires val == a && \valid(val);
ensures a_init == \old(a_init) && b_init == \old(b_init); */
```
A.6 max-1.c

Input file:

```c
int r;

int findMax(int* x, int* y) {
    if(*x >= *y)
        return *x;
    else
        return *y;
}

int* a;
int* b;
int a_init;
int b_init;
/*@ 
requires \true;
ensures a_init >= b_init ==> r == a_init;
ensures b_init > a_init ==> r == b_init;
assigns r, a, b, *a, *b;
*/
int main(void) {
    a = (int*) malloc(sizeof(*a));
    b = (int*) malloc(sizeof(*b));
    *a = a_init;
    *b = b_init;
    r = findMax(a, b);
}

Inferred contract for findMax:
/*@ 
requires x == a && y == b && \separated(x, y) && \valid(x) && \valid(y);
ensures ((b_init == \result && \result - a_init >= 1) || (a_init == \result && a_init >= b_init)) && r == \old(r) && a_init == \old(a_init) && b_init == \old(b_init) && \old(a) == a && \old(b) == b;
*/
```
Input file:

```c
int* r;

void findMax(int* x, int* y, int* max) {
    if(*x >= *y)
        *max = *x;
    else
        *max = *y;
}

int* a;
int* b;
int a_init;
int b_init;
/*@ requires \true;
ensures a_init >= b_init ==> *r == a_init;
ensures b_init > a_init ==> *r == b_init;
assigns r, a, b, *r, *a, *b; */
int main(void) {
    a = (int*) malloc(sizeof(*a));
    b = (int*) malloc(sizeof(*b));
    r = (int*) malloc(sizeof(*r));
    *a = a_init;
    *b = b_init;
    findMax(a, b, r);
}
```

Inferred contract for `findMax`:

```c
/*@ requires max == r && x == a && y == b && \separated(x, y) && \separated(x, max) && \separated(y, max) && \valid(x) && \valid(y) && \valid(max);
ensures (\old(b_init) - \old(a_init) >= 1 || \old(a_init) >= \old(b_init)) && a == \old(x) && b == \old(y) && r == \old(max) && \old(a_init) == a_init && \old(b_init) == b_init; */
```
A.8 multadd-1.c

Input file:

```c
void multiplyByTwo(int* num) {
    *num = *num * 2;
}

void addTwoNumbers(int* a, int* b, int* result) {
    *result = *a + *b;
}

int* a;
int* b;
int* result;
int a_init;
int b_init;
/*@ 
requires \true;
ensures *a == a_init * 2;
ensures *result == *a + b_init;
assigns a, *a, b, *b, result, *result;
*/
int main(void) {
    a = malloc(sizeof(*a));
    b = malloc(sizeof(*b));
    result = malloc(sizeof(*result));
    *a = a_init;
    *b = b_init;
    multiplyByTwo(a);
    addTwoNumbers(a, b, result);
}
```

Inferred contract for `multiplyByTwo`:

```c
/*@ 
requires num == a && \separated(num, b) && \separated(num, result ) && \separated(b, result) && valid(num) && valid(b) && \valid(result);
ensures \old(num) == a && \old(b) == b && \old(result) == \old(result) && a_init == \old(a_init) && b_init == \old(b_init) && \separated(num, b) && \separated(num, result) && \separated(b, result) && valid(num) && valid(b) && \valid( result) && *num == 2\old(*num);
*/
```

Inferred contract for `addTwoNumbers`:

```c
/*@ 
requires a == a && b == b && result == result;
ensures \old(a) == a && \old(b) == b && \old(result) == result && \old(a) == \old(a) && \old(b) == \old(b) && \old(result) == \old(result) && a_init == \old(a_init) && b_init == \old(b_init) && *result == \old(a) + \old(+b);
*/
```
A.9 multadd-2.c

Input file:

```c
void addNumbers(int* x, int* y, int* result) {
    *result = *x + *y;
}

void multiplyNumbers(int* x, int* y, int* result) {
    *result = *x * *y;
}

int* a;
int* b;
int* c;
int* result1;
int* result2;
int a_init;
int b_init;
int c_init;

/*@ 
ensures *result1 == a_init + b_init;
ensures *result2 == a_init * b_init;
ensures *c == b_init + c_init;
assigns a, *a, b, *b, c, *c, result1, *result1, *result2;
*/
int main(void) {
    a = (int*) malloc(sizeof(*a));
    b = (int*) malloc(sizeof(*b));
    c = (int*) malloc(sizeof(*c));
    result1 = (int*) malloc(sizeof(*result1));
    result2 = (int*) malloc(sizeof(*result2));
    *a = a_init;
    *b = b_init;
    *c = c_init;
    addNumbers(a, b, result1);
    multiplyNumbers(a, b, result2);
    addNumbers(b, c, c);
}
```

Inferred contract for `addNumbers`:

```c
/*@ 
requires \old(a) && b == \old(b) && c == \old(c) && result1 == \old(result1) && result2 == \old(result2) && a_init == \old(a_init) && b_init == \old(b_init) && c_init == \old(c_init) && *result == \old(*x) + \old(*y);
*/
```

Inferred contract for `multiplyNumbers`:

```c
/*@ 
requires x == a && y == b && result == result2 && \separated(x, y) && \separated(x, result) && \separated(x, c) && \separated(y, result1) && \separated(y, result) && \separated(y, c) && \separated(result, c) && \separated(result, result1) && \separated(c, result1) && \valid(x) && \valid(y) && \valid(result) && \valid(c) && \valid(result1);
*/
```
ensures \old(x) == a && \old(y) == b && \old(result) == result2 && \old(x) == \old(a) && \old(y) == \old(b) && c == \old(c) && result1 == \old(result1) && \old(result) == \old(result2) && a_init == \old(a_init) && b_init == \old(b_init) && c_init == \old(c_init) && separated(x, y) && separated(x, result) && separated(x, c) && separated(x, result1) && separated(y, result) && separated(y, c) && separated(y, result1) && separated(result, c) && separated(result, result1) && separated(c, result1) && valid(x) && valid(y) && valid(result) && valid(result1) && *result == \old(*x) && \old(*y);
A.10 multadd-3.c

Input file:

```c
void addNumbers(int* x, int* y, int* result) {
  *result = *x + *y;
}

void multiplyNumbers(int* x, int* y, int* result) {
  *result = *x * *y;
}

int* a;
int* b;
int* c;
int* result1;
int* result2;
int a_init;
int b_init;
int c_init;

/*@ 
ensures *result1 == a_init + b_init + b_init;
ensures *result2 == a_init * b_init;
ensures *c == *result1;
assigns a, *a, b, *b, c, *c, result1, *result1, *result2;
*/
int main(void) {
  a = (int*) malloc(sizeof(*a));
  b = (int*) malloc(sizeof(*b));
  result1 = (int*) malloc(sizeof(*result1));
  c = result1;
  result2 = (int*) malloc(sizeof(*result2));
  *a = a_init;
  *b = b_init;
  addNumbers(a, b, result1);
  multiplyNumbers(a, b, result2);
  addNumbers(b, c, result1);  
}
```

Inferred contract for `addNumbers`:

```c
/*@ 
requires c == result1 && result == result1;
ensures \old(c) == c && a == \old(a) && b == \old(b) && result2 == \old(result2) && a_init == \old(a_init) && b_init == \old(b_init) && c_init == \old(c_init) && \old(c) == result1 && *c == \old(*x) + \old(*y);
*/
```

Inferred contract for `multiplyNumbers`:

```c
/*@ 
requires x == a && y == b && c == result1 && result == result2 &&
\separated(x, y) && \separated(x, c) && \separated(x, result) && \separated(y, c) && \separated(y, result) && \separated(c, result) && \valid(x) && \valid(y) && \valid(c) && \valid(result);
*/
```
ensures \old(c) == \old(result1) && \old(x) == a && \old(y) == b
&& \old(c) == c && \old(result) == result2 && \old(x) == \old(a)
&& \old(y) == \old(b) && \old(result) == \old(result2) &&
a_init == \old(a_init) && b_init == \old(b_init) && c_init == \old(c_init)
&& \old(c) == result1 && \old(result) && \separated(x, c) && \separated(x, result) && \separated(y, c)
&& \separated(y, result) && \separated(c, result) && \valid(x) &&
\valid(y) && \valid(c) && \valid(result) && *result == \old(*x)
) * \old(*y);
A.11 truck-1.c

Input file:

```c
int driver_distance;
int r_cab_position;
typedef struct Human {
    int distance_driven;
} Human;
typedef struct Truck {
    int x;
    struct Human driver;
} *TruckPtr;

void tick(TruckPtr t) {
    t->x++;
    t->driver.distance_driven++;
}

/*@ 
requires \true;
ensures driver_distance == 2;
ensures r_cab_position == 2;
assigns driver_distance , r_cab_position;
*/
int main(void) {
    TruckPtr r_cab = malloc(sizeof(*r_cab));
    r_cab->x = 0;
    r_cab->driver.distance_driven = 0;
    tick(r_cab);
    tick(r_cab);
    driver_distance = r_cab->driver.distance_driven;
    r_cab_position = r_cab->x;
}
```

Inferred contract for tick:

```c
/*@ 
requires \true;
ensures \old(driver_distance) == driver_distance && \old(
    r_cab_position) == r_cab_position && t->x == 1 + \old(t->x) &&
    t->driver.distance_driven == 1 + \old(t->driver).distance_driven;
*/
```
A.12 truck-2.c

Input file:

```c
int christian_distance;
int gustav_distance;
int r_cab_position;

typedef struct Human {
    int distance_driven;
} *HumanPtr;

typedef struct Truck {
    int x;
    struct Human* driver;
} *TruckPtr;

void tick(TruckPtr t) {
    t->x++;
    t->driver->distance_driven++;
}

/*@ 
requires \true;
ensures christian_distance == 2 && gustav_distance == 3;
ensures r_cab_position == christian_distance + gustav_distance;
assigns christian_distance, gustav_distance, r_cab_position;
*/
int main(void) {
    TruckPtr r_cab = malloc(sizeof(*r_cab));
    HumanPtr christian = malloc(sizeof(*christian));
    HumanPtr gustav = malloc(sizeof(*gustav));
    r_cab->x = 0;
    christian->distance_driven = 0;
    gustav->distance_driven = 0;
    r_cab->driver = christian;
    tick(r_cab);
    tick(r_cab);
    r_cab->driver = gustav;
    tick(r_cab);
    tick(r_cab);
    tick(r_cab);
    christian_distance = christian->distance_driven;
    gustav_distance = gustav->distance_driven;
    r_cab_position = r_cab->x;
}
```

Inferred contract for tick:

//No inferred contract found for tick