Adding Basic Support for Function Pointers in TriCera

Axel Bergström
Abstract

TriCera is a verification tool that encodes programs in a C-like language to a set of Constrained Horn Clauses. These clauses describe the program states that can be reached when the program is executed. A solver can then be used to check if the program is safe. Before this project TriCera did not have any support for function pointers, which are a commonly used construct in C programs.

In this report, an algorithm encoding function pointer calls is defined. The calls are encoded in a way that is similar to how other conditional control flow is encoded. For each possible callee the function pointer in a call could point to, the algorithm will generate a call to that function in the encoding together with a condition checking if the pointer currently points to the function.

This algorithm is implemented in TriCera together with support for basic function pointer declarations. Test cases are used to check that the implementation works for basic programs. The implementation currently includes support for: basic function pointer declarations, function pointer expressions involving indirection and address operators, and function pointer calls. With the new additions TriCera is now able to verify basic programs using function pointers.

Limitations in the method used to find the set of possible callees and the already existing method for automatic recursion detection prevents some programs from being encoded. Future work includes: full support for function pointer declarations, a better method for finding the set of possible callees, and a better method for automatic recursion detection.
Contents

1 Introduction .......... 6
  1.1 Problem Description .................................. 6
  1.2 Contributions of this Project .......................... 7
  1.3 Outline of this Report ................................. 8

2 Background .......... 8
  2.1 Constrained Horn Clauses .......................... 8
    2.1.1 Encoding Function Calls ........................ 9
    2.1.2 Example Program Encoding ........................ 9
  2.2 Function Pointers in C ............................. 11

3 Design of Function Pointer Call Encoding .......... 12
  3.1 Definition of Branches in CHC encoding of a Program .. 12
  3.2 Overview of Encoding ................................ 13
  3.3 The Encoding Algorithm .............................. 13
  3.4 Example Encoding of Function Pointer Call ........... 14

4 Implementation in TriCera .......... 15
  4.1 Supported Function Call Encodings .................. 16
  4.2 Supported Function Pointer Declarations ............ 16
  4.3 Generating the Possible Callee Set .................. 17
  4.4 Limitations with Automatic Recursion Detection .......... 17
  4.5 Limitations with Dead Code Detection ................. 17

5 Evaluation of the Implementation .......... 18

6 Related Work .......... 18
  6.1 Verification of Higher-Order Functional Programs .......... 18
  6.2 Function Pointer Elimination ........................ 20
  6.3 Pointer Assignment Equivalence Classes ................ 20
  6.4 Type Based Filtering of Points-to Sets ................ 21

7 Future Work .......... 21
  7.1 Evaluation of the Implementation .................... 21
  7.2 Improvements to the Encoding ....................... 22
  7.3 Other Approaches to Encoding ....................... 23

8 Conclusion .......... 23
1 Introduction

TriCera\(^1\) is a model checker for C programs developed and maintained at Uppsala University [1]. It works by encoding an input program in a C-like language to a set of Constrained Horn Clauses (CHCs). These clauses describe the program states that can be reached when the program is executed. Properties to check about the program are also included in the clauses. A CHC solver such as Eldarica\(^2\) is then used to solve the generated clauses, checking if the properties included in the encoding hold or not [2].

TriCera verifies both explicit and implicit safety properties of the input program. Explicit properties can be added by inserting \texttt{assert(...)} statements in the program. Implicit properties include, for example, type safety of pointer dereferences and bounds checking for array accesses. TriCera will output \texttt{SAFE} if all safety properties hold and \texttt{UNSAFE} otherwise.

1.1 Problem Description

The main problem this thesis attempts to solve is that TriCera currently does not have any support for function pointers. There are two main goals that need to be achieved to solve this problem.

1. Define a CHC encoding for function pointer calls in C programs.
2. Implement this encoding into TriCera.

Support for function pointers would be beneficial since they are a commonly used construct in C programs. Some common function pointer usage patterns are described in [3]. It should be possible to add support for encoding function pointer calls since TriCera already is able to encode direct function pointer calls. Calling a function pointer should have the same effect as calling the function the pointer is pointing to.

Listing 2 contains a small example of a C program using function pointers. The program stores 3 to the variable \(a\), stores 2 to the variable \(b\), and then finally asserts that \(a + b = 5\). It behaves very similarly to the program in Listing 1 which does not use function pointers. Running TriCera on the code in Listing 1 outputs \texttt{SAFE} while running TriCera on the code in Listing 2 outputs an error. TriCera is not able to encode the program in Listing 2 despite the program’s simple behavior.

The code in Listing 2 should be sufficient to encode the function pointer call in the program. In the \texttt{select_one} function, the call to the \texttt{select} function pointer will either call the \texttt{min} function or the \texttt{max} function. The program also includes the implementations, or the function bodies, of both the \texttt{min} and the \texttt{max} functions. When the \texttt{select} function pointer points to the \texttt{min} function it is therefore possible to use the encoding of the \texttt{min} function to encode the call’s behavior. When the pointer points to the \texttt{max} function it is possible to use the encoding of the \texttt{max} function instead. The behavior of the function pointer call can be encoded since the implementations of both functions the pointer can point to are available.

\(^1\)https://github.com/uuverifiers/tricera
\(^2\)https://github.com/uuverifiers/eldarica
More generally, it is possible to encode the behavior of a function pointer call if the following two conditions hold: 1. It is possible to create a list containing all functions the pointer could point to. 2. The implementations of all functions in the list are available. If these two conditions hold for all function pointers in a program then it is possible to encode the behavior of all function pointer calls in that program. This should make it possible to encode at least some programs that use function pointers.

1.2 Contributions of this Project

This project has two main contributions.

1. An encoding of function pointer calls is defined.

   The basic idea is to encode function pointer calls in a way that is similar to how other conditional control flow is encoded. This can be done since, when a function pointer is called, the body of a function is executed if and only if the function pointer points to that function.

2. Basic support for function pointers has been implemented in TriCera.

   The encoding of function pointer calls has been implemented into the verification tool TriCera. Support for some but not all types of function-pointer calls has been implemented.

Listing 1: Program computing the value of \( \max(2, 3) + \min(2, 3) \) with an assert checking that the result is 5.

```c
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int main()
{
    int a = max(2, 3);
    int b = min(2, 3);
    assert(a + b == 5);
}
```

Listing 2: Program computing the value of \( \max(2, 3) + \min(2, 3) \) using function pointers with an assert checking that the result is 5.

```c
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int min(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int select_one(int a, int b, int (*select)(int, int))
{
    return select(a, b);
}

int main()
{
    int a = select_one(2, 3, max);
    int b = select_one(2, 3, min);
    assert(a + b == 5);
}
```
pointer declarations has been added. The list of all available functions
taking the correct number of parameters is used for the list of functions
each pointer could point to. Some simple test cases to verify that the
implementation works correctly for basic function pointer usage.

1.3 Outline of this Report

The rest of this report is organized in the following way: Section 2 introduces
constrained Horn clauses and how they can be used to encode programs. It also
introduces function pointers and function calls in C. Section 3 explains the de-
sign of the function pointer call encoding implemented for this thesis. Section 4
describes some more details about the implementation of the encoding in Tri-
Cera. Section 5 describes the evaluation of the implementation and presents
the results from the evaluation. Related work is discussed in Section 6. This
includes methods for call graph analysis, a method for encoding programs in a
higher-order functional language, and a tool for eliminating function pointers
from the source code of C programs. Section 7 describes another approach to
add support for function pointers and also describes some possible improvements
to the encoding implemented here. Section 8 summarizes the contributions of
this thesis and the results of the evaluation.

2 Background

This section gives a short explanation of how TriCera encodes programs using
Constrained Horn Clauses (CHCs). Section 2.1 summarizes the description of
how TriCera encodes programs from [1]. Section 2.1.1 briefly explains the
two ways in which TriCera can encode calls. Section 2.1.2 describes some
examples of encodings using CHCs. A more general introduction to using CHCs
for verification can be found in for example [4]. Section 2.2 briefly explains
function pointers in the C language. It is assumed that the reader is familiar
with first-order logic and the basics of the C language.

2.1 Constrained Horn Clauses

A CHC is a formula \( \forall x : C \land B_1 \land B_2 \land \cdots \land B_n \rightarrow H \) where \( x \) is a vector
of all first-order variables occurring in the CHC, \( B_1 \ldots B_n \) are applications of
first-order predicates to first-order terms, \( H \) is false or an application of a first-
order predicate to first-order terms, and \( C \) is a constraint over some theories.
Here, a term is defined to be either a variable, a constant, or an application
of a function to other terms. Informally, the constraint \( C \) can be described as
a formula evaluating to true or false in a language the solver can understand.
CHCs are often written as \( H \leftarrow B_1 \land B_2 \land \cdots \land B_n \land C \) leaving the universal
quantification implicit.

TriCera encodes C programs by converting them to sets of CHCs. In the
encoding each predicate corresponds to the set of possible states at a program
location. All predicates are uninterpreted. An interpretation of an uninterpreted
predicate corresponds to a set of possible states for a program location. To
show that a set of CHCs is satisfiable a solver has to find interpretations for all
predicates that make all the CHCs true. The encoding is done in a way that ensures that the program is safe if and only if the set of CHCs are satisfiable.

The entry point of a program is encoded by the CHC \( P(\ldots) \leftarrow true \). Control flow is encoded by CHCs of the form \( P_1(\ldots) \leftarrow P_0(\ldots) \land C \). In this case the constraint \( C \) can encode, for example, conditional control flow or updates to variables. An assertion of the property at the program location corresponding to \( P_n \) is encoded by two clauses, \( false \leftarrow P_n(\ldots) \land \neg C \) and \( P_{n+1}(\ldots) \leftarrow P_n(\ldots) \land C \) where the constraint \( C \) is true if and only if the property holds. The first clause ensures that the set of all CHCs is unsatisfiable if \( C \) does not hold at the program location. The second clause ensures that the control flow can continue as normal if \( C \) holds. Function calls can be encoded by inlining or by using contracts consisting of pre- and post-conditions for the function. Some example encodings are provided in Section 2.1.2.

2.1.1 Encoding Function Calls

TriCera can encode function calls both by using inlining and by using contracts. If inlining is used, then clauses encoding the callee will be added to call sites in the encoding of the caller. If contracts are used, then pre- and post-conditions are used to encode the call instead. Clauses ensuring that the postconditions follow from the preconditions are also added in this case. Pre- and post-conditions can either be specified using ACSL or be generated by the solver. The encoding of contracts and calls to contract functions in TriCera is for example described in [5]. All examples in this report use inlining to encode function calls.

Inlining cannot be used to encode recursive functions, contracts have to be used instead. If a recursive call is inlined, then this will cause another recursive call to be inlined. This process will never terminate. A function does not necessarily have to call itself for this problem to occur. If a function \( f \) calls another function \( g \) that calls \( f \) the problem will still occur. In general, the call chain leading to another call to \( f \) can have any length.

2.1.2 Example Program Encoding

The program in Listing 3 computes \( \sum_{i=1}^{10} i \) and asserts that it is equal to 55. Figure 1 contains an example encoding of this program using CHCs. The predicates \( P_0 \) to \( P_6 \) represent different program locations marked by comments in the listing. Clause 1 encodes the entry point of the program. Clauses 2, 3, 6, and 7 encode updates to program variables. Clause 7 also encodes the control flow at the end of the loop body, the program should check the loop condition again. Clauses 4 and 5 encode the loop condition. Clauses 8 and 9 encode the assertion. Figure 2 contains a graph representing the encoding. Finding interpretations of the predicates that make Clauses 1 to 9 true will show that the program is safe.

Another example can be found in Figure 3 which contains an encoding of the program in Listing 4. The functions min and max have been encoded by inlining. Clauses 11 to 15 encode the call to the max function. Clause 11 encodes the values of the arguments. Clause 12 and 13 encode the if statement condition. Clause 14 and 15 encode the return value. Clauses 17 to 21 encode the call to the min function in a similar way. The rest of the clauses in the encoding are
int main ()
{
    // P0
    int sum = 0;
    // P1
    int i = 1;
    // P2
    while (i <= 10) {
        // P3
        sum += i;
        // P4
        i += 1;
    }
    // P5
    assert (55 == sum);
} // P6

Listing 3: Program which asserts that $\sum_{i=1}^{10} i$.  

\begin{align}
    P_0() &\leftarrow \text{true} \quad (1) \\
    P_1(sum') &\leftarrow P_0() \land sum' = 0 \quad (2) \\
    P_2(sum, i') &\leftarrow P_1(sum) \land i' = 1 \quad (3) \\
    P_3(sum, i) &\leftarrow P_2(sum, i) \land i \leq 10 \quad (4) \\
    P_4(sum, i) &\leftarrow P_2(sum, i) \land \neg(i \leq 10) \quad (5) \\
    P_4(sum', i) &\leftarrow P_3(sum, i) \land sum' = sum + i \quad (6) \\
    P_2(sum, i') &\leftarrow P_4(sum, i) \land i' = i + 1 \quad (7) \\
    P_6() &\leftarrow P_5(sum, i) \land 55 = sum \quad (8) \\
false &\leftarrow P_5(sum, i) \land \neg(55 = sum) \quad (9)
\end{align}

Figure 1: CHC encoding of program in Listing 3.

Figure 2: Graph representation of CHCs in Figure 1. The nodes in the graph correspond to predicates in the clauses and the edges correspond to the clauses themselves. Each edge is labeled with a variable update $\text{var} := \text{newval}$ if the constraint in the clause is encoding an assignment or labeled with a boolean condition if the constraint encodes conditional control flow.
int max(int a, int b) {
  if (a > b)
    return a;
  else
    return b;
}

int min(int a, int b) {
  if (a < b)
    return a;
  else
    return b;
}

int main() {
  int a = max(2, 3);
  int b = min(2, 3);
  assert(a + b == 5);
  return b;
}

Figure 3: CHCs encoding of program in Listing 4.

similar to ones found in the previous example. Again, finding interpretations of the predicates making Clauses 10 to 24 true will show that the program is safe.

2.2 Function Pointers in C

This section gives a short introduction to function pointers and function calls in the C language. Basic usage of function pointers is explained by using an example. Function designators and the way they often get converted to function pointers is explained. Callees in function calls are also discussed briefly.

Basic Usage. Say that there is a function int f(int x). This function can be called by providing an argument f(2). It can also be assigned to function pointer variables that have a compatible type signature. The statement int (*fp)(); declares a function pointer fp which returns an integer. The types of the parameters of the function pointer can also be included in the declaration. Then the declaration would be int (*fp1)(int); or int (*fp2)(int x);. A pointer to the function f can be assigned to the fp variable. This is true for all three ways of declaring the fp variable. After f has been assigned to fp, the variable fp can then be called using the same syntax as for regular functions fp(2). This will call the function f.

Function Designators. A function designator is an expression which returns an object of a function type when evaluated. In most contexts, function designators are converted to expressions evaluating to objects of a function pointer
type. The exceptions to this are when the function designator is the operand of the `sizeof`, `_Alignof`, or `&` operators [6, p.54].

One consequence of this is that there are many different combinations of address operator and indirection operators that should evaluate to the same result. Say that there is a function \( f \). Then the expression \( f \) is a function designator since it evaluates to the function \( f \). The expressions \( &f \) and \( f \) should both evaluate to pointers to the same function. In the first expression, the function designator \( f \) does not get converted to a function pointer since it is the operand of the \( & \) operator. In the second expression, the function designator \( f \) is converted to a function pointer since it is not the operand of a special operator.

Applying the indirection operator \( * \) to a function pointer will usually evaluate to the same function pointer. Consider what would happen when evaluating \( *f \). The subexpression \( f \) is a function designator and will evaluate to a pointer to \( f \). Applying the \( * \) operator to the function pointer would then evaluate to a function. But this means that \( *f \) also is a function designator and will therefore be evaluated to a function pointer instead. This means that evaluating \( *f \) will give the same result as evaluating \( f \). The expressions \( **********f \) and \( *&*\&**&f \) will also evaluate to the same result as \( f \).

Evaluating Callees. The callee expressions in calls are always expressions evaluating to function pointers [6, p.81]. Say that there is a function \( f \) with one integer parameter and a function pointer \( fp \) pointing to \( f \). Then this means that in the call \( f(2) \) the function designator \( f \) is evaluated to a function pointer. In the call \( fp(2) \) the variable \( fp \) is already a function pointer. The callee expression is evaluated to a function pointer in both cases. This means that there is no fundamental difference between calling functions and function pointers in the C language.

3 Design of Function Pointer Call Encoding

This section presents a method for encoding function pointer calls using constrained Horn clauses. TriCera already has support for direct function calls. The goal of this design is to add support for function pointer calls. Subsection 3.1 defines the concept of a branch in the encoding. Subsection 3.2 gives an overview of the main idea behind the encoding algorithm, which encodes function pointer calls as multiple branches. Subsection 3.3 defines the encoding algorithm for function pointer calls. Finally, Subsection 3.4 presents an example encoding of a program containing a function pointer call.

3.1 Definition of Branches in CHC encoding of a Program

Informally, a branch in a CHC encoding of a program is a set of predicates and clauses encoding a part of a program that is only executed when some condition holds. For example, an encoding of an if-statement would have two branches. One of them corresponds to the part of the program that is executed when the condition in the if-statement is true. The other one corresponds to the part of the program that is executed when the condition is false.

This concept can also be defined in a more formal way:
• We define a state predicate as a predicate corresponding to the set of possible states at a program location in the encoding of a program.

• If $C$ is a CHC and $P$ is a state predicate, we say that $C$ follows from $P$ if and only if an application of $P$ is in the antecedent of $C$. Using the definitions in Section 2.1, this is equivalent to: $C$ follows from $P$ if and only if some $B_i$ is an application of $P$.

• If $C_1$ and $C_2$ are two CHCs, we say that $C_2$ follows from $C_1$ if and only if the head of $C_1$ is a state predicate $P$ and $C_2$ follows from $P$.

• If $C_1, C_2, \ldots, C_n$ are CHCs such that $C_i$ follows from $C_{i-1}$ for all $2 \leq i \leq n$, then we also say that $C_n$ follows from $C_1$.

• If $P$ is a state predicate from which multiple CHCs follow, we say that $P$ is a branching point.

• If $P$ is a branching point and $C$ is a CHC following from $P$, we say that the set containing $C$ and all CHCs following from $C$ is a branch. We also say that $C$ is the first clause of that branch.

3.2 Overview of Encoding

This idea for the encoding is visualized in Figure 5. The graph in Figure 4 represents a direct function call. There are some states before the function call, some states inside the call to the function ‘F’, and some states after the function call. In the graph these states are represented by the nodes ‘Before’, ‘Inside F’, and ‘After’. Each of these nodes could represent multiple states, but there is only one node in the graph for simplicity. The important thing to note here is that the control flow, which is represented by the edges in the graph, is linear. There are no branches. The graph in Figure 5 instead represents a function call through a function pointer that points to one of the functions ‘F’, ‘G’, or ‘H’. Here, the control flow has multiple branches, one for each possible callee. The encoding should be created in such a way that exactly one of these branches is taken each time the function is called.

3.3 The Encoding Algorithm

Algorithm 1 contains pseudocode encoding a function pointer call. The algorithm generates one branch for each possible callee. The guard for a branch is an expression that is true if and only if the expression denoting the called function is evaluated to the callee of the branch. The constraint in the first CHC for each branch will be modified to also include the branch guard. At most one branch guard will be true since each branch has a different callee. To ensure that a branch is always taken, a CHC asserting that one branch guard is true is added. A post-call predicate corresponding to the program state immediately after the function call returns will also be created. Finally, one CHC connecting the state predicate containing the return value of the callee of each branch will be added. This ensures that the post-call predicate will contain all possible return values from the call in any interpretation that proves the safety of the encoded program.
The algorithm assumes that generate_clauses(callee, arg₁, arg₂, . . .) is a function generating clauses encoding a direct function call. It should return a non-empty set of clauses encoding a call to callee with arg₁, arg₂, . . . as arguments. This encoding should have an unique entry clause. It should also have an unique exit predicate containing the return value of the call. The function entry(...) should give the unique entry clause to the encoded function call when applied to this set. The function exit(...) should give the unique exit predicate. It is not required that the generate_clauses(...) function is defined for all possible combinations of callees and arguments. For example if callee is a function with two parameters and only one argument is provided to the call then generate_clauses(...) could be left undefined for callee. This function makes it possible to use the same algorithm for encoding function pointer calls with different encodings of direct function calls.

The possible_callees variable contains a set of all possible callees. Reducing the size of this set will reduce the number of generated clauses. This set should be generated such that generate_clauses(callee, arg₁, arg₂, . . .) is defined for all callees in possible_callees with the provided arguments. All potential callees should also be included in this set.

### 3.4 Example Encoding of Function Pointer Call

Listing 5 contains a program which calls the function add1 through the function pointer r. Figure 6 contains a CHC encoding of the program based on Algorithm 1. Clauses 25 and 26 encode the program entry and the assignment to the function pointer r. The assertion in Algorithm 1 is encoded by Clause 27. In this example the possible_callees set contains the two functions add1 and add2. Two branches are therefore created for the call to r. Clauses 28 and 29 encode the call to add2 in one of the branches. Clauses 30 and 31 encode the call to add1 in the other branch. Clause 28 has the conjunct \( f = \text{add2} \) in the constraint which corresponds to the added conjunct target = callee in the algorithm. A corresponding conjunct also exists in Clause 30. Clauses 32 and 33 encode the return from the function pointer call. The predicate \( P_b(f, ref_p) \)
Algorithm 1  Algorithm generating clauses for a function call. The variable $fp_{clauses}$ will contain the encoding after the algorithm terminates.

**target** := first-order variable representing the callee

**possible callees** := set of all possible callees

$fp_{clauses}$ := set of CHCs asserting that $target \in possible_{callees}$

$P_{post-call}$ := new predicate containing a variable for the return value

for $i \in \{1..num\_args\}$ do
    $arg_i$ := first-order term representing argument $i$
end for

for callee $\in possible\_callees$ do
    callee clauses := generate_clauses(callee, arg_1, arg_2, ...)
    change constraint $C$ of entry(callee clauses) to $C \land target = callee$
    final clause := $P_{post-call} \leftarrow exit(callee\_clauses)$
    $fp_{clauses}$ := $fp_{clauses}$ $\cup$ callee clauses $\cup$ \{final clause\}
end for

| Function pointer declarations | Basic function pointers supported. It is assumed that all functions return integers. |
| Function pointer expressions   | Indirection and address operators supported. |
| Function pointer calls         | Supported with some limitations. There might be some limitations with how the set of possible callees is generated. Limitations exist with recursion. |

Table 1: Summary of supported features in the TriCera implementation of the design from Section 3.

corresponds to the $P_{post-call}$ predicate in the algorithm. Finally, Clauses 34 and 35 encode the assignment to $x$ and the exit from $main$ respectively.

4 Implementation in TriCera

An encoding of function pointer calls based on Algorithm 1 described in Section 3 has been implemented into TriCera. Subsection 4.1 describes the two different ways TriCera now can use to encode call expressions. Subsection 4.2 describes what types of function pointer declarations are supported in the implementation. Support for applying the indirection ($\ast$) and address ($\&$) operators to function pointers has also been implemented. Subsection 4.3 describes how the possible callees set is generated in the implementation and the limitations of this approach. Subsection 4.4 describes some limitations caused by the limited automatic recursion detection in the preprocessor. Subsection 4.5 describes some limitations caused by the preprocessor removing functions that are not called directly. Table 1 contains a summary of what features are supported.
```c
int add1(int a) {
    return a + 1;
}

int add2(int b) {
    return b + 2;
}

int main() {
    int (*f)() = add1;
    int x = f(2);
}
```

Listing 5: Program that calls the add1 function through the pointer f.

Figure 6: CHC encoding of the program in Listing 5 based on Algorithm 1. The possible callees set for the call contains the two functions add1 and add2.

### 4.1 Supported Function Call Encodings

After the additions made for this thesis, call expressions can be encoded in two different ways: As direct function calls and as function pointer calls. Direct function calls were already supported before this thesis. Support for function pointer calls based on the encoding in Section 3 has been added as a part of this thesis. Call expressions are encoded as direct function calls when the callee expression is a function name and as function pointer calls otherwise.

**TRICERA** can encode direct function calls using two different methods: by using inlining and by using contracts specifying pre- and post-conditions. In the implementation of Algorithm 1 the `generate_clauses(...)` function can use both inlining and contracts.

### 4.2 Supported Function Pointer Declarations

The implementation created for this thesis only supports basic function pointer declarations and assumes that all function pointers return integers. Declarations that are not supported include, for example, function pointer pointers and function pointer arrays. There are no extra limitations for the parameters of function pointers. Function pointer struct fields and function pointers as arguments are supported, but have the same limitations as other function pointer declarations. The reason for these limitations is that the syntax tree currently used in **TRICERA** is quite difficult to work with which slowed down development. With some more development time it would be possible to add support for more cases.
4.3 Generating the Possible Callees Set

In the implementation the possible callees set is selected as all available functions taking the same number of parameters as the number of arguments that are given to the call. To test this method TriCera was modified to always encode call expressions as function pointer calls. After this change all of TriCera’s existing regression tests still gave the correct SAFE/UNSAFE output. This method of generating the set is chosen for the implementation since it is simple to implement and ensures that the regression tests give the correct output for the modified version of TriCera.

There are however some potential problems with this implementation. Say that there is a function \( f(A \ a) \) and that there is a function pointer call \( fp(b) \) where \( b \) has type \( B \). The current implementation will generate clauses assigning \( b \) to \( a \). An attempt to generate clauses like this can occur even if it is impossible for \( fp \) to point to \( f \). This could be a problem if objects of type \( B \) cannot be assigned to objects of type \( A \). Removing this limitation is not entirely trivial because of the semantics of assignments in C and some implementation details in the existing codebase and is left as future work because of time limitations.

4.4 Limitations with Automatic Recursion Detection

The TriCera preprocessor can automatically detect functions which directly call themselves as recursive. It cannot detect recursion through other functions or through function pointers. Only functions which directly call themselves are automatically marked as contract functions by the preprocessor.

The problem with inlining not being possible for recursive functions described in Section 2.1.1 can occur more easily in programs that use function pointers. To avoid this problem, the user sometimes has to manually mark some functions as contract functions. It is not always obvious which functions to mark as contract functions. When encoding a function pointer call, clauses encoding a call will be generated for each function in the possible callees set. If clauses are generated for a function because of this we will say that there is a potential call to that function. Recursion through potential calls can cause the same problem as recursion through normal calls.

4.5 Limitations with Dead Code Detection

The TriCera preprocessor will detect functions that are not called directly as dead code and remove them before encoding the program. This limits the number of programs with function pointers that can be encoded. It is possible for a function to be called through a pointer without being called directly. A program can also use a pointer to a function without calling it. In both of these cases it is necessary to ensure that the preprocessor does not remove functions, even if they are not called directly. This can be accomplished by disabling the preprocessor, but this will lead to other limitations. A better solution would be to update the method the preprocessor uses to detect which functions should be removed.
5 Evaluation of the Implementation

Test cases have been implemented to test the basic features of the implementation works correctly. These features include evaluation of indirection and address operators, calls through function pointers, reassignment of function pointers, and function pointers as parameters. Test cases include the programs in Listings 6, 7, 8, and 9 and also the program in Listing 2 from the introduction. It is necessary to run TriCera without the preprocessor to verify these programs because of the limitation discussed in Section 4.5. The updated version of TriCera is now able to give the correct SAFE/UNSAFE output for these test cases.

Listing 6 contains a program testing the indirection and address operators for function pointers. Implementing these operators for function pointers is not entirely trivial. This is because of how function designators and function pointers behave. A short description of this behavior can be found in Section 2.2. TriCera correctly identifies this program as SAFE.

Function pointer variables and function pointer calls have been tested. Listing 7 contains a program testing function pointer assignment and calls through function pointer variables. Listing 8 extends this test case with reassignments. Listing 9 contains a call to a function pointer variable that has not been assigned any value. TriCera correctly identifies Listings 7 and 8 as SAFE and the Listing 9 as UNSAFE.

Finally, Listing 2 from the introduction has also been used as a test case. This case tests function pointers as parameters to other functions. The control flow in this case is also more complex than in the other cases presented here because of the multiple return statements. TriCera correctly identifies this program as SAFE.

6 Related Work

Some other approaches to verifying code using function pointers or higher-order functions are discussed in this section. A method for verifying code in a higher-order functional language is discussed in Subsection 6.1. A tool for eliminating function pointers from the source code of C programs is also discussed in Subsection 6.2.

Having access to a call graph could be useful for reducing the set of possible callees for each call and for better automatic recursion detection. Some work introducing and evaluating different methods for this are described in Subsections 6.3 and 6.4. Section 7.2 briefly describes how.

6.1 Verification of Higher-Order Functional Programs

Another approach to verifying C code with function pointers would be to automatically discover pre- and post-conditions for each function pointer. This could remove the need for multiple branches in each function pointer call. An implementation of this could for example be based on [7], which shows how to automatically find pre- and post-conditions for functions in a simple higher-order functional language.
Listing 6: Program used to test the evaluation of the indirection and address operator on function pointer operands.

Listing 7: Program used to test assignment of function pointers to variables and calls through function pointer variables.

Listing 8: Program used to test assignment of function pointers to variables, calls through function pointer variables, reassignment of function pointer variables, and calls to function pointer variables after reassignment.

Listing 9: Program used to test calls to an unassigned function pointer.
The paper defines a refinement type system for the simple higher-order language and a function that generates verification conditions for a program in this language. The refinement type system includes constraints on variables, arguments, and return values of functions. It is known to be sound in the sense that if types can be found for a program then that program will never fail. It is also known to be incomplete, there are programs that never fail where types cannot be found. The function generating verification conditions for a program is based on this type system. If the verification conditions for a program are satisfiable then there exists types for all variables and functions. The types will include constraints on all function arguments and return values. These constraints will describe pre- and post-conditions for the functions. The verification conditions generated for a program can be converted to a set of Horn clauses.

6.2 Function Pointer Elimination

A tool for converting C programs which use function pointers into C programs with equivalent functionality but without any function pointer calls is presented in [8]. The tool uses three main steps to accomplish this: the input program is converted to clang bitcode, the clang bitcode is analyzed to produce over-approximations of values for function pointer variables, and finally the over-approximations are used to modify the original C program producing a new one without any function pointer calls.

Some experiments showing that the tool is sufficiently efficient for real-world programs are also presented in the paper. The tool can for example eliminate all function pointers from the git project in 13 minutes and 47 seconds, and the OpenSSL project in 16 minutes and 13 seconds.

6.3 Pointer Assignment Equivalence Classes

An algorithm for partitioning the object names appearing in a program into equivalence classes is presented in [9]. Object names are expressions created from some combination of variable names, heap names\footnote{Heap names are of the form heap\textsubscript{id} where \textit{id} is an identifier assigned to each heap allocation statement}, structure field accesses, and pointer dereferences. They refer to memory locations or addresses of memory locations. The equivalence classes are represented by the PE relation (Pointer-related-assignment-induced-Equality relation) where each object name belongs to exactly one class. If two object names can have the same value then they belong to the same equivalence class. It is shown that the algorithm has an almost linear time complexity of $O(n\alpha(n,n))$.

The precision that can be achieved by using the PE relation for call graph construction in C programs with function pointers is analyzed in [3]. For each function pointer call $(\star \textit{fp})(\ldots)$ the authors assume that every function in the same PE equivalence class as $\star \textit{fp}$ could be called. They give some reason for why one could expect this method to be inaccurate. The method is flow-insensitive, it does not consider control flow or the order of program segments. It is also context-insensitive and does not differentiate between different contexts from which a function is called. Finally, the method is also symmetric, it will infer both that $x$ can contain the previous value of $y$ and that $y$ can contain the previous value of $x$ from the assignment $x = y$. The algorithm is however able
to differentiate between structure fields. Despite these reasons to expect the method to be inaccurate they found that it generated the true call graph for some realistic C programs.

To evaluate the precision of the method the authors use a set of 8 publicly available realistic C programs which use function pointers. The PE relation was computed for each program and used to construct a call graph. By doing some manual work they were able to show that the true call graph was found by the algorithm for all 8 tested programs. All calls that can occur but no calls that cannot occur were included in the constructed call graph. Explanations for why this method is sufficient for some common function pointer usage patterns are given in the paper. They conclude that this inexpensive analysis may provide sufficient precision for call graph construction.

6.4 Type Based Filtering of Points-to Sets

A method for using type information to improve call graph precision in C programs using function pointers is presented in [10]. The points-to set of a function pointer refers to the set of all functions the pointer could point to. To improve call graph precision type information is used to filter the points-to set yielding a subset. Two methods for filtering are introduced. Strong filtering assumes that programs follow the ANSI standard. Weak filtering makes weaker assumptions. Experiments show that both filtering methods can be an effective way of reducing the number of edges in the call graph. Strong filtering is however shown to be unsafe in that it can sometimes remove calls that can occur from the call graph. The experiments do not show that weak filtering is unsafe but the authors ‘do not dispute’ the claim made in [11] stating that the use of type information for this purpose is unsound.

In [11] the authors claim that the number and types of parameters cannot be safely used to refine the set of functions that could be invoked from a function pointer call site. The reason given for this is that C permits both passing a variable number of arguments to functions and type casting.

7 Future Work

Two types of future work are discussed here. Subsection 7.1 discusses some future work related to the evaluation of the implementation. Subsection 7.2 discusses some possible improvements that could be made to the design and the implementation of the encoding. Subsection 7.3 discusses another approach to encoding programs with function pointers.

7.1 Evaluation of the Implementation

It would be beneficial to have a more in-depth evaluation of the implementation. Having some real-world test cases would be especially useful to check if the implementation is able to verify real-world code. I did make an attempt to use SV-Comp [12] instances\(^4\) for the evaluation. The attempt failed since I could not manage to find any instances that had function pointer calls, could be parsed and encoded by TRIÇERA and where the function pointer calls mattered for

\(^4\)Available online at https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
the verification. Some more work is needed in order to be able to use SV-Comp instances to evaluate the implementation. If this work is to be done, the first step should probably be to figure out why TriCera is not able to parse and encode these instances.

The implementation could also be compared with other approaches to verifying programs using function pointers. One other approach is to use function pointer elimination to convert the input programs into programs without support for function pointers. One method for doing this is presented in [8], which is discussed in Section 6.2. A comparison can then be made between the new version of TriCera created for this project and the old version of TriCera together with function pointer elimination.

7.2 Improvements to the Encoding

Work can also be done to address some of the issues described in Section 4. Solving these issues could allow TriCera to verify more programs.

- A wider range of function pointer declarations would be beneficial to increase the number of programs that can be encoded by TriCera.
- It would also be beneficial to not assume that all function pointers point to functions returning integers since this assumption is clearly false.
- A better method for generating the possible callees set could also be beneficial. This could remove the potential issue described in Section 4.3. A smaller set of possible callees would also give a smaller encoding of the program. This might make solving easier, but this should be tested using empirical measurements.
- Updating the method used by the preprocessor to detect which functions to remove from the input could increase the number of programs that can be verified.

Having access to a call graph for the program being encoded could also enable some improvements. Say that there is a call graph where the nodes \( F_1, F_2, \ldots, F_n \) are functions in the program and there is a directed edge \((F_a, F_b)\) if \( F_a \) can call \( F_b \). Then the set of possible callees for every function pointer call in a function \( F_a \) should be a subset of the set of functions \( F_b \) such that there is an edge \((F_a, F_b)\). This observation can be used to refine the set of possible callees. Extra edges in the graph can be allowed, if there is an edge \((F_a, F_b)\) that does not imply that \( F_a \) can call \( F_b \). The work presented in [3] and the work presented in [10], both of which are discussed in Section 6, describe some methods that might be useful for generating such graphs.

If the call graph is used to refine the sets of possible callees, then the following observation could be used to address the inlining issue described in Section 4.4: Each inlining of a function in the encoding corresponds to one edge in the call graph. This is still true if all incoming edges are removed from nodes corresponding to contract functions. The inlining problem can only occur if there is at least one cycle in the graph created by removing all such edges. Marking functions as contract functions until there are no cycles in this graph will get rid of the inlining issue.
7.3 Other Approaches to Encoding

A CHC encoding for C programs with function pointers could also be based on the encoding presented in [7] and briefly discussed in Section 6.1. An encoding based on this approach could automatically find pre- and post-conditions for function pointer variables, instead of for function variables in a higher order function language. Adapting the refinement-type based method presented in the paper to the C language might require a lot of work. To reduce the amount of work required, it might be possible to use some other method to generate the verification conditions while still generating pre- and post-conditions for each function pointer variable. Future work could investigate methods to generate pre and post-conditions for function pointer variables.

8 Conclusion

Before this project TriCera did not have any support for function pointers. Support for function pointer calls based on Algorithm 1 has been implemented in TriCera. Support for function pointer variable declarations has also been added. The algorithm will create one branch for each possible function that a function pointer could point to. Some basic test cases for the encoding have been created. The implementation works for these basic cases but more comprehensive testing would be beneficial.

There are some limitations with the implementation due to time limitations. Only basic function pointer declarations are supported. It is assumed that all function pointers point to functions returning integers. The method used to find the set of possible callees when a function pointer is called is very simple. The automatic recursion detection is also limited at the moment. These problems are all implementation problems. None of these are fundamental problems with the encoding of function pointer calls.

Even though there are some limitations with the current implementation TriCera now has basic support for function pointers. A CHC encoding for function pointer calls has been defined and implemented into TriCera. The two main goals of this project have been achieved.

References


