Optimizing Stack Allocation for the CakeML Compiler

MSc Thesis in Computer Science – Algorithms, Languages, and Logic.

RICHARD AHLIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Optimizing Stack Allocation for the CakeML Compiler

RICHARD AHLIN

Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Optimizing Stack Allocation for the CakeML Compiler
RICHARD AHLIN

© RICHARD AHLIN, 2022.

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Gerardo Schneider, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Optimizing Stack Allocation for the CakeML Compiler
RICHARD AHLIN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis concerns the formal verification of an optimization in relation to memory stack allocation and deallocation operations in the context of tail calls, in an intermediate language of the CakeML compiler. The thesis explores and provides a discourse on the implementation and associated proof of a compiler phase in a verified compiler in which the semantics of each phase must be correct with respect to the previous one. CakeML is a functional programming language based on Standard ML, and has a compiler that is fully verified. A primary focus in this thesis is put on the formal proof, where the optimization and its implementation merely serve as a foundation.

Keywords: Verification, Formal Methods, Compiler Optimizations, Functional Programming, CakeML.

Acknowledgements

Firstly, I would like to especially thank my supervisor Magnus Myreen for the continuous support and guidance he has given throughout this whole project. I would also like to thank the CakeML team for providing help when needed. Lastly I would like to thank my examiner Gerardo Schneider for showing interest and providing valuable feedback to this report.

Richard Ahlin, Gothenburg, July 2022

Contents

Contents **9**

1 Introduction **11**
1.1 Motivation . . . 11
1.2 Tail Calls . . . 12
1.3 The Optimization . . . 13
1.4 Research Question . . . 14
1.5 Goals and Challenges . . . 14
1.6 Contributions . . . 14
1.7 Methodology . . . 14
1.8 Limitations . . . 15
1.9 Structure of this Report . . . 15

2 Background **17**
2.1 CakeML . . . 17
2.1.1 The StackLang Intermediate Language . . . 17
2.2 The HOL4 Interactive Theorem Prover . . . 18

3 Optimization Example **21**

4 Implementation **23**
4.1 Preliminaries . . . 23
4.2 Implementation Details . . . 23
4.2.1 compile . . . 24
4.2.2 create_raw_eps . . . 25
4.2.3 opt_code . . . 26

5 Verification Proof **29**
5.1 StackLang Semantics . . . 29
5.1.1 Definition of evaluate . . . 30
5.1.2 Handling Divergence . . . 35

5.2 Proving Semantics Preservation 36
5.2.1 Example: Verification of a Simple Optimization 38
5.2.2 Three Useful Techniques in Verification 38
5.3 The Stack Allocation Optimization Verification 39
5.3.1 Specifications for code_rel 41
5.4 Procedure for Constructing the Proof 42

6 Related Work **45**

7 Conclusion **47**

Bibliography **49**

1
Introduction

1.1 Motivation

Faulty software exists everywhere and is encountered by people regularly. Often these problems are surmountable, and does not cause major problems in the everyday lives of people because they are not critical. Other times, software bugs are not benevolent, and may in fact wreak havoc at the tail end. Take for instance the Pentium FDIV bug costing Intel $475 million USD in damages, being caused by a missing lookup table entry. Another example is the Ariane V space rocket that exploded shortly after its launch, caused by an integer overflow and resulting in $1 billion USD in damages [1].

Formal verification is an area concerned with the construction of rigorous mathematical proofs about software or hardware systems. The verification is carried out in relation to a formal specification, and once the proof is complete, the software (or hardware) system is proven to behave according to the specification. It is impossible to formally verify every aspect of a system (and its interaction with the world), so any formally verified system may still have errors, but they are guaranteed not to be found in the scope where the formal proofs apply. Altogether, a formal verification is an immensely valuable asset to have, and is ranked higher than any kind of traditional testing.

Software in the modern age is developed by writing computer programs as source code in a programming language, and then compiling it to machine code with a compiler. The software developer is concerned with keeping his or hers source code bug free, but once the source code reaches the compiler, the control of program behavior, or program *semantics*, is lost to the compiler. The compiler should not under any circumstances change the external behavior of a program, the so called *observational semantics*. Any change in observational semantics would amount to a different program than the developer intended to, meaning that the compiler would introduce a bug into the program, indicating a flawed compiler. In other words; a correct compiler does preserve observational semantics in the compilation. Since all higher-level languages eventually compile to machine code, an important topic is to try hard in keeping the compiler correct, since otherwise

this could potentially affect all programs being compiled by it. Therefore, applying formal verification on the development of a compiler would amount to a much higher degree of accuracy than otherwise could be obtained.

The main purpose of a compiler is to translate the source code into machine code, and doing so while preserving the observational semantics. However, it is often very desired for a compiler to optimize the program behavior by changing the internal semantics, as to make it as efficient as possible. Every such optimization, or *optimizing code transformation*, is a distinct pass in the compilation process. For verified compilers, every such pass needs to be verified independently to prove that it by itself does not alter the external behaviour. Once every pass in compiler has been verified, one can called the compiler a *fully formally verified* compiler.

CakeML is a functional programming language based on a substantial subset of Standard ML [2] and has a fully formally verified compiler. The CakeML compiler features several intermediate languages, and performs optimizations along the way [3]. This thesis explores the implementation, and more importantly, the formal proof, of a compiler pass concerning an optimization for the CakeML compiler.

1.2 Tail Calls

Within a function in the source code of a programming language, if the last statement is a function call, then it is called a *tail call*. This placement of a function call has a fortunate implication: there is no need to return to the function once the function call has been made, since there are no further statements or expressions to be executed in the function. This means that there is no need to keep the data in memory that the original function needed. Instead, that stack frame can be discarded at the moment of the tail call. This in essence is a *tail call optimization*, being especially important in cases of recursive tail calls, where these can be acted upon like iterations in a loop memory wise, rather than a linearly growing amount of stack memory.

Note: The optimization in this thesis is not a tail call optimization. Instead, this thesis' optimization proceeds from an already implemented tail call optimization and optimizes stack memory operations in relation to it.

Examples of tail calls, that are also tail recursive, can be seen in Figure 1.1 (Standard ML), Figure 1.2 (C) and Figure 1.3 (assembly).

```
fun return0 0 = 0
  | return0 n = return0 (n - 1)
```

Figure 1.1: Recursive tail call in Standard ML.

```
int return0(int n) {
  if(n == 0)
    return 0;
  return return0(n - 1);
}
```

Figure 1.2: Recursive tail call in C.

```
foo:
  call bar
  jmp foo
```

Figure 1.3: Recursive tail call in assembly. Normal function call denoted by *call* and tail call denoted by *jmp*.

1.3 The Optimization

The optimization presented in this thesis begins at the already implemented tail call optimization in the CakeML compiler, and more specifically in the StackLang intermediate language of the compiler. The tail call optimization is implemented such that prior to any tail call is made, the stack frame of the caller function is removed. Then, the amount of stack memory needed by the targeting function of the call is allocated as a first action before its execution.

1. Deallocation.
2. Tail call.
3. Allocation.

In the context of tail calls, this is a recurring pattern and features redundant stack operations. Instead of two distinct stack memory operations, they can be merged to one memory operation reflecting the net memory difference in the end. Deallocating 5 memory units and then allocating 7 memory units can be replaced by one allocation of 2 memory units. In practise, this simple optimization reduces two instructions to only one (or zero if the memory amounts are the same) dealloaction or allocation instruction. This does not change the observation behavior of the code, but results in a faster program execution.

1.4 Research Question

The CakeML project as an organization has a higher goal of presenting the viability of as *realistic* of a compiler as possible, that is also formally verified. Every optimization in the compiler makes it more realistic, as it increases the likelihood of it being used in a real setting, and thus being able to compete with non-verified compilers. The research question of this thesis relates to the higher goal of the organization, but narrows the focus to present the viability of the formal verification procedure of a single optimization.

1.5 Goals and Challenges

This thesis' primary goal is to conduct a verified formal proof of the stack allocation optimization described in Section 1.3, which states the perseverance of observational semantics. Formal proofs in this manner can become long and tedious, and might be difficult for beginners, as being the case of this thesis. This will be the most challenging part of the thesis. Fortunately, the Computer Science Master's programme at Chalmers University of Technology has given mandatory and voluntary courses related to formal proofs which have given an introduction to the area of formal verification. Aside from the verification, the implementation part of this thesis is simply standard functional programming, and is seen as an easier task.

1.6 Contributions

This thesis makes the contributions of an implementation of an optimization, a verified formal proof, and a discourse on the formal verification of code transformations within verified compilers.

1.7 Methodology

The work of this thesis began with writing the optimization, and performing some traditional tests on it. Once it was deemed to work as expected based on the tests, the formal verification process began. Inside the theorem prover, where the formal verification is performed, a statement in logic was formally expressed to give meaning to what it formally meant for the optimization to preserve observational semantics. Then, as a step-by-step process, the proof was built backwards from the statement, turning it into a theorem.

Lastly, the verified optimization could be implemented into the CakeML compiler (not in the scope of this thesis). All of the work, including implementing

the optimization, was carried out inside the HOL interactive theorem prover (see Section 2.2).

1.8 Limitations

The optimization presented in this thesis will improve execution time in CakeML programs featuring tail calls, but likely not by much. It is outside the scope of this thesis to perform benchmarks measuring the performance boost.

1.9 Structure of this Report

This report follows with a background for the work in Chapter 2, a practical example on how the optimization works in Chapter 3, the implementation of the optimization in Chapter 4, the formal proof in Chapter 5, related work in Chapter 6, and lastly a conclusion of this thesis' work in Chapter 7.

2
Background

This chapter provides the necessary background for the thesis. CakeML the programming language and the compiler are first described in Section 2.1, with StackLang the intermediate language in Section 2.1.1. Lastly, HOL4 the interactive theorem prover (the environment in which CakeML is being developed) is described in Section 2.2.

2.1 CakeML

CakeML is a functional programming language based on a substantial subset of Standard ML, and features data types, pattern-matching, higher-order functions, exceptions, modules and more [4]. The CakeML compiler is fully formally verified, meaning that the semantics of the source code are proven to be preserved under the compilation to machine code. The compiler currently targets the hardware architectures: x86-64, ARMv6, ARMv8, MIPS-64, RISC-V and Silver ISA. The compilation of a CakeML source code is established through the eight intermediate languages of the CakeML compiler (can be seen in Figure 2.3). The optimizing code transformation of this thesis takes place in the StackLang intermediate language.

2.1.1 The StackLang Intermediate Language

StackLang is an imperative programming language used under the compilation process in the CakeML compiler as an intermediate language (IL), with semantics defined in a functional big-step style [5]. The StackLang IL supports a concrete implementation of the memory stack [6] along with features commonly found in imperative languages such as while loops, if statements and function calls.

Because the IL supports a model of the stack, some stack operations are also available; memory allocation and deallocation (denoted StackAlloc and StackFree in the StackLang IL) being the important operations of this thesis. The entire set of StackLang instructions, along with its abstract syntax, can be seen in the *prog* data type in Figure 2.2. Referenced data types to it are explained Table 2.1.

The *prog* data type is recursively defined and StackLang programs are represented by it. One (nested) instance of the *prog* data type represents a function, and the collection of such functions a whole program. A program can be seen as a tree which is held together by some instructions that can hold further instructions, for example while loops, if statements, and a special instruction called Seq with an only purpose of holding instructions, presenting a path of program execution by first evaluating the first instruction, and then the second. This StackLang program representation can nest to an arbitrarily depth. An example of a StackLang function issuing a tail call can be seen in Figure 2.1.

Name	Meaning	Name	Meaning
α	Machine word length	inst	Assembly instruction
num	Natural number	store_name	Global variable name
cmp	Boolean expression	reg_imm	Register name or immediate constant
string	String	option	Option type

Table 2.1: Data types referenced by the *prog* data type.

```
Seq (StackAlloc 5)
  (Seq (...) (Seq (StackFree 5) (Call None (INL 7) None)))
```

Figure 2.1: A function in the StackLang IL making a tail call.

2.2 The HOL4 Interactive Theorem Prover

The HOL4 interactive theorem prover provides an implementation of higher-order logic, as well as a suite for proving theorems defined in that logic. The higher-order logic supports typed functions, data type declarations and pattern matching, and can thus be used as a purely functional programming language. Software (e.g. the CakeML compiler) are written in that logic and later proved correct using certain tools, many of which are provided by the theorem prover.

The verification occurs by first specifying a *proof goal* and then manually applying so called *proof tactics* until the goal has been proved, making it a theorem. The proof tactics are functions themselves that are able to deconstruct the goal into several sub-goals which are further broken down and proved separately using more tactics in a recursive fashion. At some point, the sub-goals are simple enough for the theorem prover to solve them by itself.

Once program code in the higher-order logic has been written (and verified), it is possible to synthesize semantically equivalent CakeML code using an automatic and fully verified procedure [7], making the compiler able to bootstrap itself.

```
α prog =
   Skip
 | Inst (α inst)
 | Get num store_name
 | Set store_name num
 | Call ((α prog × num × num × num) option) (num + num)
 ((α prog × num × num) option)
 | Seq (α prog) (α prog)
 | If cmp num (α reg_imm) (α prog) (α prog)
 | While cmp num (α reg_imm) (α prog)
 | JumpLower num num num
 | Alloc num
 | Raise num
 | Return num num
 | FFI string num num num num num
 | Tick
 | LocValue num num num
 | Install num num num num num
 | CodeBufferWrite num num
 | DataBufferWrite num num
 | StackAlloc num
 | StackFree num
 | StackStore num num
 | StackStoreAny num num
 | StackLoad num num
 | StackLoadAny num num
 | StackGetSize num
 | StackSetSize num
 | BitmapLoad num num
 | Halt num
```

Figure 2.2: The StackLang *prog* data type.

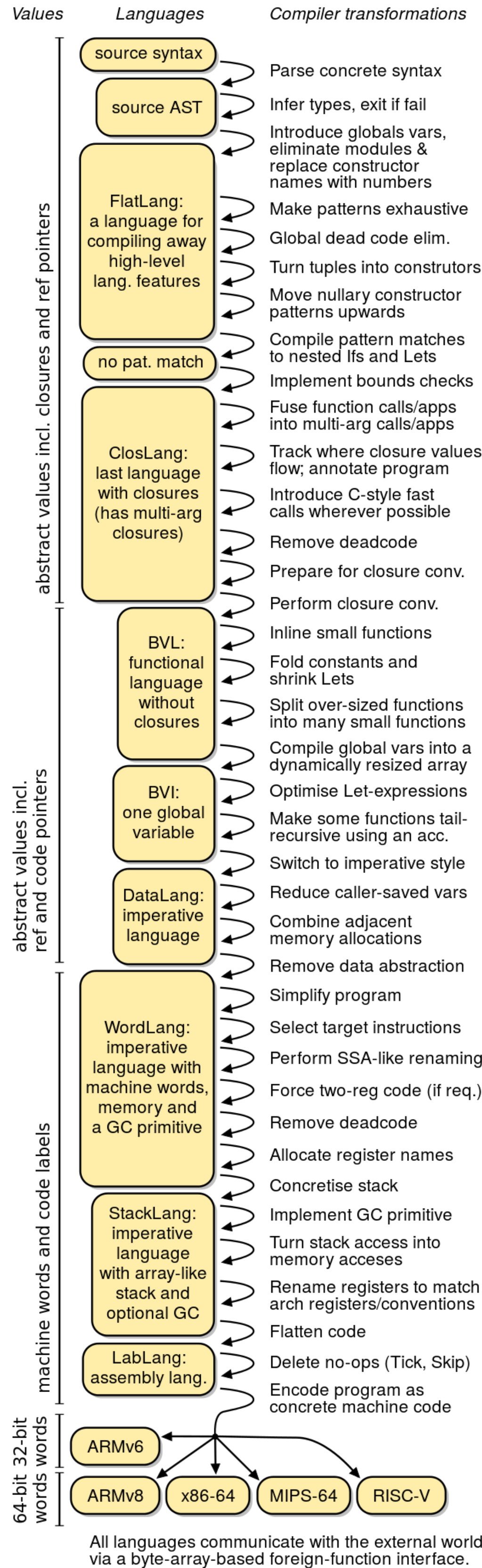

Figure 2.3: The various stages of compilation in the CakeML compiler, featuring all ILs. The StackLang IL can be seen in the bottom part of the figure. Source: cakeml.org

3

Optimization Example

This chapter provides a practical example behind the technical workings of the optimization of this thesis, but is presented as pseudocode. This first piece of code shows a non-optimized program.

```
1  foo:
2     StackAlloc 5
3     ...
4     StackFree 5
5     Jump foo
6     ...
7     StackFree 5
8     Jump bar
9     ...
10
11 bar:
12    StackAlloc 7
13    ...
```

Defined are two functions, `foo` and `bar`, with each function body starting with a memory allocation (`StackAlloc`) according to the memory requirements of that function. Arbitrary code (triple dots) then proceeds in `foo` until a tail call occurs (`Jump`), with a removal of the allocated stack frame (`StackFree`) just before.

A recursive tail call to `foo` is made at line 5, thus deallocating the same amount of memory that will be allocated on re-entrance; being semantically equivalent to omitting both of them.

A tail call from `foo` to `bar` is made at line 8, thus deallocating 5 memory units followed by allocating 7 memory units at the start of `bar`. This is semantically equivalent to transforming the deallocation into an allocation of 2 memory units (representing the net change in memory requirements), and omitting the original allocation instance in the beginning of `bar`.

The approach presented in this thesis optimizes the program according to this methodology, and includes the introduction of so called *raw entry points*, one for each original function. The resulting optimized program looks like the following:

```
1  foo:
2     StackAlloc 5
3     Jump foo_raw
4
5  foo_raw:
6     ...
7     Jump foo_raw
8     ...
9     StackAlloc 2
10    Jump bar_raw
11    ...
12
13 bar:
14    StackAlloc 7
15    Jump bar_raw
16
17 bar_raw:
18    ...
```

Virtually all StackLang functions start with a memory allocation. All code that follows can be made into a separate function, here called the raw entry point of the function (`foo_raw` and `bar_raw` in the optimized program). The memory allocation stays in the original function (`foo` and `bar`), with an additional tail call to the raw entry point. Whenever an ordinary function call is issued, the plain function is called, allocating necessary memory, and continues to the code of the raw entry point. But when a tail call is made, the raw entry point of the wanted function is called directly. The only other change is that the memory deallocation prior to the tail called is changed to reflect the net difference of memory requirements between the calling function and its target.

At line 8 in the non-optimized program where `foo` makes a tail call to `bar`, the `StackFree 5` followed by `StackAlloc 7` is made into a single `StackAlloc 2` in the optimized program. At line 5 in the non-optimized program where `foo` makes a recursive tail call, the deallocation is omitted entirely.

4

Implementation

This chapter describes the implementation of the stack allocation optimization, implemented as a set of functions in the interactive theorem prover HOL4. The implementation consists of a main function, compile, that uses nine helper functions. When using this optimization pass in the CakeML compiler, a StackLang program is sent as an argument to compile, which returns the optimized program.

4.1 Preliminaries

Inside the CakeML compiler, a compiling program in the StackLang IL is represented as a list of tuples with two values, each tuple representing a function of the program. The first value of a tuple represents the name of the function (simply named with an even natural number). The second value of a tuple is the StackLang code of that function, an instance of the *prog* datatype. Even natural numbers are used as a naming convention of the functions in order to make room for the raw entry points which are given names of odd numbers, and always a single increment above the number of the original function. An abbreviated StackLang program as represented in the compiler, before the optimization, can be seen in Figure 4.1.

```
[(0, Seq (StackAlloc 3) (...)), (2, Seq (StackAlloc 7) (...)), ...]
```

Figure 4.1: Example of a program in the StackLang IL before the optimization.

4.2 Implementation Details

The bulk of the implementation consists of the main function called compile, a helper function to it called create_raw_eps, which together prepare for the actual optimization, which is lastly done in a function called opt_code. These three functions will be presented in their own section, together with smaller helper functions they use.

4.2.1 compile

Compiling CakeML programs passes through different ILs in the compiler, until they eventually transform into StackLang code, and at some point during this stage, they will be optimized as specified in this thesis. When that happens, the main function compile will be called, with the StackLang program representation as an only argument to it.

$$
\begin{array}{l}
\mathsf{compile}\ \mathit{stack_lang_prog} \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ \mathit{tree}\ =\ \mathsf{mem_reqs}\ \mathit{stack_lang_prog}\ \emptyset \\
\quad \mathsf{in} \\
\qquad (\mathsf{create_raw_eps}\ \mathit{stack_lang_prog}\ \mathit{tree},\mathit{tree})
\end{array}
$$

The memory requirements of the stack frames of the different functions in a StackLang programs are first extracted using the helper function mem_reqs, and are then stored in a tree called *tree*. The function create_raw_eps is then called with the StackLang program and the memory tree as arguments.

Helper function

The function compile uses one helper function: mem_reqs, which in turn uses a helper function called get_alloc.

1. mem_reqs The mem_reqs function is used to extract the stack memory requirements of the different functions in the compiling program.

$$
\begin{array}{l}
\mathsf{mem_reqs}\ []\ \mathit{tree} \stackrel{\text{def}}{=} \mathit{tree} \\
\mathsf{mem_reqs}\ ((\mathit{name},\mathit{prog})::\mathit{xs})\ \mathit{tree} \stackrel{\text{def}}{=} \\
\quad \mathsf{mem_reqs}\ \mathit{xs}\ (\mathsf{insert}\ \mathit{name}\ (\mathsf{get_alloc}\ \mathit{prog})\ \mathit{tree})
\end{array}
$$

The function recursively steps through the StackLang program, one tuple (function) at a time. For each function, the name and extracted memory amount (retrieved using get_alloc) are added to the tree until all functions have been added.

$$
\begin{array}{l}
\mathsf{get_alloc}\ (\mathsf{Seq}\ (\mathsf{StackAlloc}\ \mathit{mem})\ y) \stackrel{\text{def}}{=} \mathit{mem} \\
\mathsf{get_alloc}\ \mathit{other} \stackrel{\text{def}}{=} 0
\end{array}
$$

The function get_alloc pattern matches the StackLang code on the initial stack memory allocation, and simply returns the amount.

4.2.2 create_raw_eps

The create_raw_eps function creates and returns a new list of functions (StackLang program) that is the final optimized program. That list is also created recursively by issuing calls to helper functions, including the more substantial function opt_code.

$$
\begin{aligned}
&\textsf{create_raw_eps}\ []\ \mathit{tree} \stackrel{\text{def}}{=} []\\
&\textsf{create_raw_eps}\ ((\mathit{name},\mathit{prog})::\mathit{xs})\ \mathit{tree} \stackrel{\text{def}}{=}\\
&\quad \textsf{let}\ \mathit{raw_ep}\ =\ \textsf{create_raw_ep}\ \mathit{prog};\ \mathit{raw_name}\ =\ \mathit{name}+1\\
&\quad \textsf{in}\\
&\qquad (\mathit{name},\textsf{strip_fun}\ \mathit{prog}\ \mathit{raw_name})::\\
&\qquad\quad (\mathit{raw_name},\textsf{opt_code}\ \mathit{raw_ep}\ \mathit{tree})::\\
&\qquad\qquad \textsf{create_raw_eps}\ \mathit{xs}\ \mathit{tree}
\end{aligned}
$$

The raw entry point of a function is created by calling create_raw_ep, which removes the initial memory allocation and stores the remaining code in *raw_ep*, as well as giving it a name in *raw_name*. Then, this new list of functions (the optimized program) is augmented with two functions: the function with memory allocation and tail call, and the raw entry point.

The function with memory allocation and tail call is created by calling strip_fun on the original function code, which removes all code except for the aforementioned allocation. A tail call to the raw entry point is then added. The second function, the raw entry point, is augmented onto this new list by calling opt_code on it, which does the heavy lifting in the optimization of this thesis.

Helper functions

The function create_raw_eps uses two helper functions: create_raw_ep and strip_fun.

1. create_raw_ep

$$
\begin{aligned}
&\textsf{create_raw_ep}\ (\textsf{Seq}\ x\ y) \stackrel{\text{def}}{=} y\\
&\textsf{create_raw_ep}\ \mathit{other} \stackrel{\text{def}}{=} \mathit{other}
\end{aligned}
$$

The function create_raw_ep discards the first instruction in the code of a function, which always is a memory allocation. The rest of the function code is simply returned.

2. strip_fun

$$
\begin{aligned}
&\textsf{strip_fun}\ (\textsf{Seq}\ x\ y)\ \mathit{raw_name} \stackrel{\text{def}}{=} \textsf{Seq}\ x\ (\textsf{Call None}\ (\textsf{INL}\ \mathit{raw_name})\ \textsf{None})\\
&\textsf{strip_fun}\ \mathit{other}\ \mathit{raw_name} \stackrel{\text{def}}{=} \textsf{Call None}\ (\textsf{INL}\ \mathit{raw_name})\ \textsf{None}
\end{aligned}
$$

The function strip_fun preserves the memory allocation of the code of a function, discards the rest of the code and introduces a tail call to the raw entry point of the same function.

4.2.3 opt_code

The opt_code function makes changes to existing tail calls in the function code, as well as modifying the deallocations prior to the tail calls.

```
opt_code prog tree =
  case prog of
  | Seq x y  ⇒
    (case (dest_StackFree x,dest_Call_NONE_INL y) of
      (Some curr_mem,Some name)  ⇒
        case lookup name tree of
          Some dest_mem  ⇒
            Seq (new_mem curr_mem dest_mem)
                (Call None (INL (name + 1)) None))
        | None  ⇒  Seq x y
    | other  ⇒  Seq (opt_code x tree) (opt_code y tree)
  | If c r ri x′ y′  ⇒
    If c r ri (opt_code x′ tree) (opt_code y′ tree)
  | While c′ r′ ri′ x′′  ⇒
    While c′ r′ ri′ (opt_code x′′ tree)
  | other  ⇒  other
```

The opt_code function recursively steps through the function code one instruction at a time. It pattern matches on Seq, If and While, but merely propagates the optimization to the nested instructions in the two latter cases. For Seq, it further pattern matches on instances where the first instruction is a memory deallocation and the second a tail call. Otherwise, it also merely propagates the optimization.

When opt_code accurately pattern matches on a tail call, the deallocation amount is extracted using the helper function dest_StackFree, as well as the function name of the target of the tail call using the helper function dest_Call_NONE_INL. The memory requirement of the target function is then extracted from the memory tree using lookup. Finally, a new instance of Seq is constructed, where the original deallocation is replaced by a call to new_mem together with the two memory amounts (calling function and target function) as arguments. It returns an allocation or deallocation representing the net memory difference between the functions. The original tail call is redirected to the raw entry point of the targeting function. Thus, the original Seq instance is replaced with the new modified version.

Helper functions

The function opt_code uses three helper functions: dest_StackFree, dest_Call_NONE_INL and new_mem.

1. dest_StackFree

$$\mathsf{dest_StackFree}\ (\mathsf{StackFree}\ m) \stackrel{\mathsf{def}}{=} \mathsf{Some}\ m$$
$$\mathsf{dest_StackFree}\ \mathit{other} \stackrel{\mathsf{def}}{=} \mathsf{None}$$

The function dest_StackFree returns the memory amount of a deallocation instruction.

2. dest_Call_NONE_INL

$$\mathsf{dest_Call_NONE_INL}\ (\mathsf{Call\ None}\ (\mathsf{INL}\ \mathit{name})\ \mathsf{None}) \stackrel{\mathsf{def}}{=} \mathsf{Some}\ \mathit{name}$$
$$\mathsf{dest_Call_NONE_INL}\ \mathit{other} \stackrel{\mathsf{def}}{=} \mathsf{None}$$

The function dest_Call_NONE_INL returns the name of the targeting function in a tail call instruction.

3. new_mem

$$
\begin{array}{l}
\mathsf{new_mem}\ \mathit{curr_mem}\ \mathit{dest_mem} \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if}\ \mathit{dest_mem} < \mathit{curr_mem}\ \mathsf{then} \\
\quad\quad \mathsf{Seq\ Tick}\ (\mathsf{StackFree}\ (\mathit{curr_mem} - \mathit{dest_mem})) \\
\quad \mathsf{else\ if}\ \mathit{curr_mem} < \mathit{dest_mem}\ \mathsf{then} \\
\quad\quad \mathsf{Seq\ Tick}\ (\mathsf{StackAlloc}\ (\mathit{dest_mem} - \mathit{curr_mem})) \\
\quad \mathsf{else\ Tick}
\end{array}
$$

The function new_mem is used to generate either an allocation or deallocation instruction with the specific memory amount reflecting the net memory requirement difference.

5

Verification Proof

This chapter describes the verification proof of the stack allocation optimization in this thesis. The proof entails a verification that the observational semantics have been preserved under the optimizing transformation; meaning that the semantics of a StackLang code prior to the transformation equates the semantics of the code after the transformation.

The semantics of the StackLang intermediate language are first described in Section 5.1. Following that, a description of the general procedure in conducting verification proofs is given in Section 5.2, starting with an example of verifying a simple optimization in Section 5.2.1. Then, three common techniques used for formal verification proofs in the CakeML compiler are presented in Section 5.2.2. Lastly, the verification proof for the optimization in this thesis is explained in Section 5.3.

5.1 StackLang Semantics

In the same way as most intermediate languages in the CakeML compiler, the StackLang IL is making use of a functional big-step semantics as its operational semantics, a style arguably well suited for compiler verification [5]. The functional style uses a recursive interpreter function, in CakeML called `evaluate`. The interpreter gives semantics to individual expressions, but as a consequence of it being recursively defined, and the fact that StackLang programs are constituted of nested expressions, the interpreter does also give low-level semantics to entire programs. Defining the operational semantics in a recursive style with an interpreter function leads to verification proofs regarding program semantics that can be carried out by induction over the interpreter function. When this succeeds, it results in theorems stating something about the observable semantics of the underlying programming language.

5.1.1 Definition of evaluate

As input to the interpreter function evaluate, a StackLang program and a *state* is given. The state is of a record type, and is used to supervise properties related to the evaluate evaluation, including any side effects the evaluation might have (since these side effects might influence the evaluation further down the line), which may become a part of the observational semantics of the program. Some of these properties make up a static configuration, while other properties are dynamic and might be changed during the evaluation. The state can essentially be seen as an environment surrounding the evaluation of a program. An extract from the interpreter function evaluate can be seen in Figure 5.1 and is further explained in the next subsection of Section 5.1.1, and the state record can be seen in Figure 5.2.

It is dependent on three parameters, and is made up from single values, lists, trees, finite maps (curved arrow with base) and functions (standard arrow), the last of which is able to capture more complex state properties. For example, the *regs* field models values kept in the processor registers, *stack* models the memory stack, and *code* holds the code of every function of a StackLang program (the program given to evaluate is essentially a single function and any other function needed is loaded from this storage). The *clock* field is used for modeling divergence in programs, and is explained in Section 5.1.2.

Combining every possible state and every possible StackLang program encapsulates every possible program evaluation that can happen (being the foundation for formal verification). An evaluation of a program together with a state returns a result and a post-state (which may be different from the pre-state because of the impurity of StackLang). The result returned from evaluate is wrapped in an option type, meaning that the result may either be something real and concrete, or a simple indication of an empty result. The different results can be seen in the result data type in Figure 5.3, and are here described.

Result (α `word_loc`)	The result returned from a StackLang function.
Exception (α `word_loc`)	Indicating an exception, carrying a value representing which exception.
Halt (α `word_loc`)	Indication that the StackLang program exited normally. (Always carrying a machine word as the result of the computation; *see paragraph below*).
FinalFFI `final_event`	The program stopped running because of a foreign function that did not respond.
Error	An error occurred during the evaluation.

The two referenced data types are *word_loc* and *final_event*. The first data type can be seen in Figure 5.4 and is defined to be either a machine word (with a size encoded by α) or the pointer to a code location Loc l_1 l_2, where l_1 corresponds to the name of a function and l_2 to a location within that function. The second data type, *final_event*, can be seen in Figure 5.5 and is used to capture information in relation to a failed FFI (foreign function interface) call.

```
⊢ (∀ s. evaluate (Skip,s) = (None,s)) ∧
  (∀ v s.
     evaluate (Halt v,s) =
       case get_var v s of
         None ⇒ (Some Error,s)
       | Some w ⇒ (Some (Halt w),empty_env s)) ∧
                          ...
  (∀ s.
     evaluate (Tick,s) =
       if s.clock = 0 then (Some TimeOut,empty_env s)
       else (None,dec_clock s)) ∧
  (∀ s c2 c1.
     evaluate (Seq c1 c2,s) =
       let (res,s1) = evaluate (c1,s)
       in
         if res = None then evaluate (c2,s1) else (res,s1)) ∧
                          ...
  ∀ v s r.
    evaluate (BitmapLoad r v,s) =
      if ¬s.use_stack ∨ r = v then (Some Error,s)
      else
        case get_var v s of
          None ⇒ (Some Error,s)
        | Some (Word w) ⇒
          if length s.bitmaps ≤ w2n w then (Some Error,s)
          else
            (None,set_var r (Word (el (w2n w) s.bitmaps)) s)
        | Some (Loc v6 v7) ⇒ (Some Error,s)
```

Figure 5.1: Selected cases from the interpreter function evaluate. There are about two dozen additional cases.

```
(α, γ, 'ffi) state = <|
  regs : num ↦ α word_loc;
  fp_regs : num ↦ 64 word;
  store : store_name ↦ α word_loc;
  stack : α word_loc list;
  stack_space : num;
  memory : α word → α word_loc;
  mdomain : α word → bool;
  bitmaps : α word list;
  compile :
   γ → (num × α prog) list → (8 word list × γ) option;
  compile_oracle :
   num → γ × (num × α prog) list × α word list;
  code_buffer : (α, 8) buffer;
  data_buffer : (α, α) buffer;
  gc_fun :
   α word_loc list ×
   (α word → α word_loc) ×
   (α word → bool) × (store_name ↦ α word_loc) →
   (α word_loc list ×
   (α word → α word_loc) × (store_name ↦ α word_loc))
   option;
  use_stack : bool;
  use_store : bool;
  use_alloc : bool;
  clock : num;
  code : α prog spt;
  ffi : 'ffi ffi_state;
  ffi_save_regs : num → bool;
  be : bool
|>
```

Figure 5.2: The state record used for StackLang.

α result =
  Result (α word_loc)
 | Exception (α word_loc)
 | Halt (α word_loc)
 | TimeOut
 | FinalFFI final_event
 | Error

Figure 5.3: The *result* data type in StackLang.

α word_loc = Word (α word) | Loc num num

Figure 5.4: The *word_loc* data type.

final_event =
  Final_event string (8 word list) (8 word list)
  ffi_outcome

Figure 5.5: The *final_event* data type.

Illustrated examples of evaluate

Here follows a few selected cases from evaluate, to illustrate how it operates. The interpreter function works by separately and independently defining semantics for all 28 StackLang instructions from the *prog* data type in Figure 2.2. Any instructions that require subexpressions to be evaluated to prior itself will issue recursive calls to evaluate on them. An entire StackLang program (can be seen a tree structure of expressions) will be evaluated by this principle until the program reaches an end.

Any time evaluate returns, a post-state is also included. This post-state might then be given to further applications of evaluate (according to the semantics) in order to keep track of the state of the evaluation. Here follows the semantic definitions for Seq, StackAlloc and If, taken from the full definition of evaluate.

Definition of Seq

$$
\begin{aligned}
&(\forall\, s\ c_2\ c_1. \\
&\quad \textsf{evaluate}\ (\textsf{Seq}\ c_1\ c_2, s) = \\
&\quad\ \ \textsf{let}\ (res, s_1) = \textsf{evaluate}\ (c_1, s) \\
&\quad\ \ \textsf{in} \\
&\quad\ \ \ \ \textsf{if}\ res = \textsf{None then evaluate}\ (c_2, s_1)\ \textsf{else}\ (res, s_1))
\end{aligned}
$$

The semantics for the Seq expression are defined such that, for all possible states, and for all possible expressions c_1 and c_2, the result of the evaluation is the result from evaluating the second expression c_2, as long as evaluating c_1 does not end the evaluation by returning a concrete value from the *result* data type in Figure 5.3. If it does, the result of the evaluation is the result from evaluating c_1.

Note here as well that the state supplied to the evaluation of c_2 is the post-state returned from evaluating c_1.

Definition of StackAlloc

$$
\begin{aligned}
&(\forall\, s\ n. \\
&\quad \textsf{evaluate}\ (\textsf{StackAlloc}\ n, s) = \\
&\quad\ \ \textsf{if}\ \neg s.\textsf{use_stack then (Some Error}, s) \\
&\quad\ \ \textsf{else if}\ s.\textsf{stack_space} < n\ \textsf{then} \\
&\quad\ \ \ \ (\textsf{Some (Halt (Word}\ 2w)), \textsf{empty_env}\ s) \\
&\quad\ \ \textsf{else (None}, s\ \textsf{with stack_space} := s.\textsf{stack_space} - n))
\end{aligned}
$$

The semantics for the StackAlloc expression are defined such that, for all possible states, and all possible natural numbers n:

1. If the state defines that the memory stack should not be used, then an error is the result.
2. If there is not enough stack space, then the result is Halt (Word $2w$) (an indication that the program ran out of stack space). The resulting post-state is the same as the pre-state but with temporary data removed.
3. If the state defines the stack to be used, and there is enough stack memory, then a decrement of size n is made on the free stack memory amount.

The semantics for the StackAlloc expression are defined in this manner because the stack operations for these purposes only work against a simple (but sufficient) model of the stack rather than towards an actual implementation of the memory stack.

Definition of If

```
(∀ s ri r1 cmp c2 c1.
   evaluate (If cmp r1 ri c1 c2,s) =
     case (get_var r1 s,get_var_imm ri s) of
       (None,v4) ⇒ (Some Error,s)
     | (Some x,None) ⇒ (Some Error,s)
     | (Some x,Some y) ⇒
       case word_cmp cmp x y of
         None ⇒ (Some Error,s)
       | Some T ⇒ evaluate (c1,s)
       | Some F ⇒ evaluate (c2,s))
```

The semantics for the If statement are defined such that, for all possible states, and all possible variables *ri* and r_1, all possible expressions c_1 and c_2, and all boolean expressions *cmp*:

1. If one or both of the values from the variables could not be retrieved, an error is the result since a lookup should not fail.
2. If both variable values could be retrieved, then the boolean expression *cmp* with the variables *ri* and r_1 is evaluated.
   (a) If the evaluation failed, then the result of the evaluation of the If statement is an error.
   (b) If evaluated to true, then the result of the evaluation of the If statement is the result of the evaluation of the first expression c_1.
   (c) If evaluated to false, then the result of the evaluation of the If statement is the result of the evaluation of the second expression c_2.

As can be seen, the semantics of the If statement does not affect the state. The pre-state is propagated to the evaluation of the subexpressions c_1 and c_2.

5.1.2 Handling Divergence

The interpreter is defined as a function in HOL4, and needs to give semantics to all StackLang programs that are syntactically valid. This presents a problem, since providing a diverging program, meaning a program that never properly terminates, should mean that the interpreter function never properly terminates either. This further means that you are unable to reason about such a program, and are unable to prove anything about it. To handle this, a natural number called *clock* is used, which resides in the state. The *clock* value is initialized before the execution of the

interpreter function and upon every step in the program evaluation which poses a risk of divergence, for instance while loops and (recursive) function calls, the *clock* value is decreased. When the clock value reaches zero, the interpreter will terminate upon next clock decrease, resulting in a TimeOut error. In order to prove a program not diverging, it is sufficient to show a single *clock* value that does not result in a TimeOut error.

5.2 Proving Semantics Preservation

While the interpreter function defines the semantics of programs at the level of single expressions, there is another function used in CakeML called semantics, which define the *top-level* semantics, being the observational semantics of an entire program. It is used in a more overarching way in proofs in the CakeML compiler. Each IL has a semantics function, producing comparable outcomes, meaning that the observational semantics of programs encoded in different ILs can be reasoned about and compared. This is when the semantics function is used in the CakeML compiler; during proofs of observational semantics preservation for programs transforming from IL_A to IL_B.

The semantics function is defined in terms of evaluate, meaning that it uses the result from evaluate and converts it into an outcome that states something about the observational semantics; something that evaluate cannot do on its own.

It is out of scope for this thesis to explain the semantics function in detail, but it is useful to know that it defines the observational semantics of an IL [6]. The outcomes of semantics are:

Fail	The semantics fail.
Diverge e	Divergence has occurred.
Terminate o e	Proper termination.

In the outcomes, e indicates the reason for termination, and o indicates the IO events produced by the program. The semantics function can be seen in Figure 5.6. Proving observational semantics preservation in an optimizing transformation is however usually done using only evaluate together with lemmas. This is because any such proof using semantics would ultimately lead to a proof about evaluate (since the former is defined with the latter).

```
⊢ semantics start s =
    let prog  =  Call None (INL start) None
    in
     if
       ∃ k.
         let res  =  fst (evaluate (prog,s with clock  :=  k))
         in
           res ≠ Some TimeOut ∧
           res ≠ Some (Result (Loc 1 0)) ∧
           (∀ w. res ≠ Some (Halt (Word w))) ∧
           ∀ f. res ≠ Some (FinalFFI f)
     then
       Fail
     else
       case
         some res.
             ∃ k t r outcome.
                 evaluate (prog,s with clock  :=  k) =
                   (Some r,t) ∧
                 (case r of
                    Result v4  ⇒  outcome = Success
                  | Exception v5  ⇒  F
                  | Halt w  ⇒
                    outcome =
                      if w = Word 0w then Success
                      else Resource_limit_hit
                  | TimeOut  ⇒  F
                  | FinalFFI e  ⇒  outcome = FFI_outcome e
                  | Error  ⇒  F) ∧
                 res = Terminate outcome t.ffi.io_events
       of
         None  ⇒
           Diverge
             (build_lprefix_lub
                (image
                   (λ k.
                        fromList
                          (snd
                             (evaluate
                                (prog,s with clock  :=  k))).
                          ffi.io_events) 𝒰(: num)))
       | Some res  ⇒  res
```


Figure 5.6: The semantics function.

5.2.1 Example: Verification of a Simple Optimization

This section presents a verification theorem for the simplest possible optimization possible, as an example. Starting with a conjecture and once turning it into a theorem, this states the perseverance of observational semantics under the optimizing transformation.

$$\vdash \mathsf{evaluate}\ (prog,s) = (res,s_1) \land res \neq \mathsf{Some\ Error} \Rightarrow \\ \quad \mathsf{evaluate}\ (\mathsf{simple_opt}\ prog,s) = (res,s_1)$$

This conjecture states:

Evaluating any syntactically valid program (*prog*) with any pre-state (*s*), resulting in a result (*res*) and a post-state (s_1), as long as *res* is not an error (which does not occur in well-typed programs), *then*,
evaluating the same program after the optimizing transformation (simple_opt *prog*) with the same pre-state (*s*), will result in the same result (*res*) and the same post-state (s_1), as the evaluation of the non-optimized program.

Once proven, this theorem states that the observational semantics have been preserved since all possible programs that undergo the optimization will generate the same result and post-state when evaluated with the same pre-state compared to the non-optimized variants of the same programs.

5.2.2 Three Useful Techniques in Verification

Often verification theorems require more statements to be true than in the simple optimization in Section 5.2.1. These statements depend on how the observational semantics are defined for the execution of a program, but the three techniques presented in this section are common in verification theorems about the perseverance of observational semantics in the CakeML compiler, and will appear in the final verification theorem for the optimization in this thesis in the next section, Section 5.3.

The state_rel Relation

state_rel *s t*

The state_rel relation defines a requirement of certain members to correspond between two states *s* and *t*. These requirements reflect what is meant by observational semantics for a particular application. For example, an optimizing transformation

might require the register values to be intact under the transformation, so they would have to correspond.

The code_rel Relation

code_rel s.code t.code

The code_rel relation defines a relation between the program code stored in two states s and t. It verifies that the optimizing transformation has performed the correct program code changes, reflecting that the t.code program code is the optimized version of the s.code program code.

Providing Extra Fuel

evaluate (opt_code $prog$,t with clock := t.clock + ck)

At times, one wants to provide extra fuel (ck) to the clock of the pre-state in the evaluation of the optimized version of the program, where ck is a natural but finite number. The reason is that the optimizing transformation might introduce elements (e.g. function calls) into the program code that results in a higher amount of clock decrements compared to the non-optimized version of the program when evaluated.

The extra finite resource hinders the evaluation of the optimized program from resulting in a TimeOut error in instances where the evaluation of the non-optimized program would terminate successfully.

5.3 The Stack Allocation Optimization Verification

This section presents the verification theorem of the stack allocation optimization. The theorem has the same structure as the theorem of the simple optimization in Section 5.2.1, and uses the three techniques presented in Section 5.2.2.

$$
\begin{array}{l}
\vdash \mathsf{evaluate}\ (prog,s) = (res,s_1) \land \mathsf{state_rel}\ s\ t \land \\
\quad \mathsf{code_rel}\ s.\mathsf{code}\ t.\mathsf{code}\ tree \land res \neq \mathsf{Some\ Error} \Rightarrow \\
\quad\quad \exists\, ck\ t_1. \\
\quad\quad\quad \mathsf{evaluate} \\
\quad\quad\quad\quad (\mathsf{opt_code}\ prog\ tree,t\ \mathsf{with\ clock}\ :=\ t.\mathsf{clock} + ck) = \\
\quad\quad\quad\quad (res,t_1) \land \\
\quad\quad\quad (\mathsf{if}\ res = \mathsf{Some\ TimeOut} \lor \exists\, r.\ res = \mathsf{Some\ (Halt}\ r)\ \mathsf{then} \\
\quad\quad\quad\quad s_1.\mathsf{ffi} = t_1.\mathsf{ffi} \\
\quad\quad\quad\ \mathsf{else\ state_rel}\ s_1\ t_1) \land \\
\quad\quad\quad \mathsf{code_rel}\ s_1.\mathsf{code}\ t_1.\mathsf{code}\ tree
\end{array}
$$

The theorem states that when:

1. Any program *prog* is evaluated with any pre-state s and it results in a result *res* and post-state s_1 in: evaluate $(prog,s) = (res,s_1)$.
2. The pre-state s is related to any state t in: state_rel s t.
3. The program code of s relates to the program code of state t and memory amounts in *tree* (from Section 4.2.1) in: code_rel s.code t.code *tree*.
4. The result *res* is not an Error.

Then, the observational semantics of a program *prog* are preserved because there exists a clock *ck* and a state t_1, such that:

5. An evaluation of the optimized program, opt_code *prog tree*, together with the pre-state t (related to s in 2) and given extra fuel *ck*, will result in the same result *res*, and with the post-state t_1 in: evaluate (opt_code *prog tree*,t with clock := t.clock + *ck*) = (res,t_1).
6. The *ffi* states of the two post-states s_1 and t_1 correspond upon the result of a TimeOut error or a Halt r, and otherwise the two post-states s_1 and t_1 relate in the state relation.
7. There is a code relation between the two post-states s_1 and t_1.

Setting the requirement of the evaluate instances to both result in *res* is an obvious requirement for semantics preservation. The requirement of the two pre-states being related under state_rel s t is done in order to ensure that the state members that influence the evaluation will correspond (i.e. setting the same 'starting point' for the two evaluate instances).

The state_rel relation used in this verification theorem can be seen in Figure 5.7 and consists of a state member correspondence of all members except *compile*, *compile_oracle*, *code_buffer* and *data_buffer*. These member are concerned with dynamic installation of new code, and do not influence the observational semantics of a program in this setting. The requirement of the two program stores, and the memory tree *tree*, to be related under code_rel s.code t.code *tree* are explained in depth in Section 5.2.2. The relation ensures that the code transformation of the optimization is correct, and that the memory tree is correct with respect to the memory requirements of the functions found in the program store. The evaluate evaluation of the optimized program is given a finite amount of extra fuel *ck*, and this is because the optimization introduces additional function calls, as explained in Section 5.2.2. If the evaluations result in a TimeOut or a Halt r, then (for the sake of observational semantics preservation) it is enough to prove a correspondence between the *ffi* states in the two post-states. In any other result that is not Error, the state_rel relation needs to hold between the post-states in order to be able to instantiate the inductive hypothesis in the proofs, which assume state_rel. Finally, it is proven that the code in the program stores are correct with respect to the optimization in code_rel s_1.code t_1.code *tree*, as proving integrity within the verification.

$$
\begin{aligned}
&\vdash \mathsf{state_rel}\ s\ t \iff \\
&\quad t.\mathsf{clock} = s.\mathsf{clock} \land t.\mathsf{regs} = s.\mathsf{regs} \land \\
&\quad t.\mathsf{stack} = s.\mathsf{stack} \land t.\mathsf{stack_space} = s.\mathsf{stack_space} \land \\
&\quad (t.\mathsf{use_alloc} \iff s.\mathsf{use_alloc}) \land t.\mathsf{bitmaps} = s.\mathsf{bitmaps} \land \\
&\quad t.\mathsf{fp_regs} = s.\mathsf{fp_regs} \land t.\mathsf{store} = s.\mathsf{store} \land \\
&\quad t.\mathsf{memory} = s.\mathsf{memory} \land t.\mathsf{mdomain} = s.\mathsf{mdomain} \land \\
&\quad t.\mathsf{gc_fun} = s.\mathsf{gc_fun} \land (t.\mathsf{use_stack} \iff s.\mathsf{use_stack}) \land \\
&\quad (t.\mathsf{use_store} \iff s.\mathsf{use_store}) \land t.\mathsf{ffi} = s.\mathsf{ffi} \land \\
&\quad t.\mathsf{ffi_save_regs} = s.\mathsf{ffi_save_regs} \land (t.\mathsf{be} \iff s.\mathsf{be})
\end{aligned}
$$

Figure 5.7: The state_rel relation definition.

5.3.1 Specifications for code_rel

The code_rel relation specifies a relation between the code stores of two states, and in this specific case also a memory tree *tree*.

The relation is denoted by: code_rel s.code t.code *tree*.

The relation verifies that the code store t.code contains the correctly optimized code of the code store s.code, with respect to the specifications of the optimization

(Chapter 3). It also verifies that the memory tree *tree* is correct with respect to the allocation amounts in the beginning of every StackLang function in s.code. The implementation of code_rel is partly built from the functions defined in the optimization implementation of Chapter 4.

$$
\begin{aligned}
&\vdash \mathsf{code_rel}\ s_code\ t_code\ tree \iff \\
&\quad \forall\, n\ x.\\
&\qquad \mathsf{lookup}\ n\ s_code = \mathsf{Some}\ x \Rightarrow \\
&\qquad\ \exists\, m.\\
&\qquad\quad \mathsf{lookup}\ n\ tree = \mathsf{Some}\ m \land \\
&\qquad\quad \mathsf{lookup}\ n\ t_code = \mathsf{Some}\ (\mathsf{strip_fun}\ x\ (n+1)) \land \\
&\qquad\quad \mathsf{lookup}\ (n+1)\ t_code = \\
&\qquad\quad\ \ \mathsf{Some}\ (\mathsf{opt_code}\ (\mathsf{create_raw_ep}\ x)\ tree) \land \\
&\qquad\quad \exists\, x_2.\ x = \mathsf{Seq}\ (\mathsf{StackAlloc}\ m)\ x_2
\end{aligned}
$$

The relation states that, for any (non-optimized) function in s.code, labeled n, and letting x be the code of the function n, then:

1. There exists a memory amount m for function n in the memory tree *tree*.
2. The code of the function n in t.code corresponds to an application of strip_fun x $(n + 1)$.
3. The code of the function *(n + 1)* in t.code corresponds to an application of opt_code (create_raw_ep x) *tree*.
4. There exists function code x_2 such that a stack allocation instruction allocating m memory units followed by the instructions of x_2 exactly corresponds to the code of the function n in s.code.

Together, 1 and 4 reflect that the memory tree *tree* is correct. 2 makes sure that the function n in t.code is the minimally stripped function of the function n in s.code. 3 makes sure that the function *(n + 1)* in t.code is the optimized raw entry point of the original function n in s.code. These statements combined express that the code t.code has been transformed correctly from s.code with respect to the optimization, and that the memory tree *tree* is correct with respect to s.code.

5.4 Procedure for Constructing the Proof

In functional big-step semantics, the interpreter function evaluate serves as a cornerstone in constructing proofs. It defines the logic (the semantics) where any further reasoning can be made. Essentially, the proof statement (see Section 5.3)

revolves around the applications of evaluate. The other parts of the proof statement simply refer to evaluate in some way, either in their arguments, result, or pre- or post-state. The function evaluate defines the complete semantics by separately and independently defining the semantics for each and every StackLang instruction (see Figure 5.1), and no more. Instructions that require the evaluation of more instructions to complete (Seq, If and While) can represent this in evaluate by recursive calls. Having the recursive function evaluate defined in this way is the reason that it is an interpreter, and can function as one on its own. Programs to be interpreted are sent as an argument to evaluate, where instructions are interpreted sequentially according to the semantics and order of the program.

Following from these principles there can be a solid manner in which to construct proofs; namely to break down the overall proof to several smaller proof cases, one for each instruction. If every individual instruction can be proven correct with respect to evaluate and the optimization opt_code for every possible state or other parameter (including recursive references to evaluate), then all these small proofs together must yield a holistic proof concerning any possible program at large. The objective then becomes to prove a goal statement for each instruction. Referring to the implementation of the optimization opt_code (Section 4.2.3), the optimization does not modify the code for most instructions. Only two instructions (If and While) propagate the optimization further but are not themselves altered and it is only Seq where the optimization modifies the code (but only for tail calls).

For the many smaller proofs, this makes them trivial. Any proof that is not Seq, If or While is easy to prove, and this because the optimization does not alter them. In the goal statements for these, the opt_code application simply acts as an identity function, with no modification made, and the evaluate instances will equate and cancel out.

The instructions If and While are trickier to prove since they involve recursive applications of evaluate, and Seq is most difficult to prove since it also contains the alteration in code. These three cases contain recursive calls to evaluate in order to encapsulate a model for any possible subexpressions, which can potentially be a very large amount. These three cases are proven using induction on the recursive function evaluate, a so called recursive induction. The induction can be carried out since code in programming languages naturally follow an inductive structure. These cases are proven using the induction hypothesis: the assumption that any subexpression is correctly compiled, and from that proving the rest of the case. The reason one can assume this correctness is that programs can never be infinitely long (even though they might run indefinitely), which means that the expressions somewhere must end. The program cannot end in an expression that is If, While or Seq, since these require more expressions. Thus, any StackLang program must end in of the other more than two dozen expressions, for which there are trivial

correctness proofs as stated further up in this section. In the context of induction, these instructions serve as proof for the base cases.

Furthermore, the written proof in HOL4 (excluding the goal statement and other pure definitions), including proofs of lemmas, reaches a bit over 600 lines in length.

6
Related Work

Formally verified compilers, assuming the verification is carried out on the same class of programming languages, will to an extent exert similar types of optimizations under the compilation process. In the CakeML compiler, especially in the StackLang IL, many optimizations will have similar verification procedures. To our knowledge, the verification of the optimization presented in this thesis, that builds on the already implemented tail call optimization, is the first of a kind.

In general, formally verified compilers are not the norm, much due to the extensive undertaking that the verification process requires, and especially so when an optimizing compiler is wanted. But there does exist work on other formally verified (and optimizing) compilers. One example is the CompCert compiler, providing a complete compilation from a large subset of the C programming language [8] into assembly code for PowerPC, ARM, RISC-V and x86 architectures[9]. It uses the Coq [10] interactive theorem prover for proving the correctness of the compiler. It is known for being the first commercially available and formally verified optimizing compiler [8]. Another formally verified compiler is the Cogent compiler [11] which introduces its own functional programming language with the same name, compiling to the C programming language. It is specifically design for low-level types of programming, and could further down be compiled to machine code using CompCert, creating a fully preserving chain of observational semantics under the compilation process.

7
Conclusion

The primary purpose of this thesis was to present the concept of formal verification on optimizing transformations within compilers as to demonstrate it being a practical and feasible option in compiler construction. This is in contrast to the fact that most compilers are not formally verified, but could arguably benefit from it, especially in applications where criticality and safety are of utmost importance. The formal verification of the optimization was successful, and could be achieved using third party tools made for verification together with an already existing infrastructure for conducting proofs within the CakeML compiler. The verification, together with tools and methodology provided in this thesis, give support to the stance that formal verification of compiler optimizations are feasible.

This thesis presented the implementation and formal verification of an optimization targeted at reducing the number of allocation and deallocation instructions in relation to tail calls in an intermediate language of the CakeML compiler, which has proofs of observational semantics preservation under the entire compilation chain, making the CakeML a fully verified compiler. The implementation was carried out in the higher-order logic of the HOL4 interactive theorem prover, where it was also proven correct. The proof formally states the perseverance of observational semantics in relation to the optimization.

Having acquired the implementation and formal proof, the optimization could be implemented into the fully formally verified CakeML compiler (out of scope for this thesis).

Bibliography

[1] T. Tuerk, "Interactive theorem proving (itp) course." Lecture, KTH Royal Institute of Technology, 2017. `https://www.kth.se/social/files/595631e456be5b96c04314d6/itp-course.pdf`.

[2] R. Milner, *The Definition of Standard ML: Revised*. MIT press, 1997.

[3] "Cakeml." `https://cakeml.org/`. Accessed: 2020-07-06.

[4] S. Owens, M. Myreen, R. Kumar, and M. Norrish, "Cakeml: A verified implementation of ml," *Principles of Programming Languages*, pp. 179–191, 2014.

[5] S. Owens, M. Myreen, R. Kumar, and Y. Tan, "Functional big-step semantics," *European Symposium on Programming Languages and Systems*, pp. 589–615, 2016.

[6] S. Owens, M. Myreen, R. Kumar, M. Norrish, Y. Tan, and A. Fox, "A new verified compiler backend for cakeml," *International Conference on Functional Programming*, pp. 60–73, 2016.

[7] S. Owens and M. Myreen, "Proof-producing translation of higher-order logic into pure and stateful ml," *Journal of Functional Programming*, pp. 284–315, 2014.

[8] D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, *et al.*, "Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler," *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*, pp. 1–9, 2018.

[9] "Compcert compiler." `https://compcert.org`. Accessed: 2021-08-22.

[10] "Coq proof assistant." `https://coq.inria.fr`. Accessed: 2021-08-22.

[11] L. O'Connor, C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. Murray, and G. Klein, "Cogent: Certified compilation for a functional systems language," 2016. arXiv:1601.05520.