Hrutvik Kanabar

Implementing and verifying a compiler optimisation for CakeML

Computer Science Tripos – Part II

King's College

May 18, 2018

Proforma

Name:	**Hrutvik Kanabar**
College:	**King's College**
Project Title:	**Implementing and verifying a compiler optimisation for CakeML**
Examination:	**Computer Science Tripos – Part II, July 2018**
Word Count:	**11990**[1]
Project Originator:	Dr. Magnus Myreen
Supervisors:	Dr. Stephen Kell & Dr. Anthony Fox

Original Aims of the Project

To implement global dead code elimination optimisations in two intermediate languages of the compiler for the open-source, functional language CakeML. These will eliminate unreachable or unnecessary code that can be removed without affecting observable behaviour from FlatLang (the first intermediate language in the compilation pipeline) and WordLang (the antepenultimate language). The WordLang optimisation should be proved to preserve the semantics of any program using the interactive theorem-prover HOL4. Lastly these optimisations should be integrated into the existing CakeML codebase, and (for the FlatLang case) evaluated using demonstrations of reduced code size.

Work Completed

Three of four core success criteria were achieved outright: the optimisation was implemented for FlatLang, both implemented and verified for WordLang, and demonstrated to remove unused FlatLang code. The final criterion was achieved as far as possible: the FlatLang optimisation is not part of the latest version of CakeML, but is a part of the latest version that contains FlatLang (§ *2.2.1*). It will be a part of the main compiler as soon as FlatLang itself is.

An extension aim of the project was also achieved: the FlatLang optimisation was proved correct.

[1]Calculated using `texcount -quiet -merge -sub=chapter -sum <.tex file>`.

Special Difficulties

Working on an active open-source project, and so coping with external timelines affecting my own work (§ *2.2.1*).

It was therefore not possible to achieve one of the success criteria: integrating the optimisation for FlatLang into the latest version of the compiler. FlatLang itself is not a part of this version, so the optimisation was instead integrated into the latest version containing FlatLang.

An (achieved) success criterion was to demonstrate the effect of the optimisation in FlatLang on example code. However, the test programs used for this would not build until May 7, 2018[2].

[2] Commit URL: `https://github.com/CakeML/cakeml/commit/52ebb2`.

Declaration

I, Hrutvik Kanabar of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: H. Kanabar

Date: May 18, 2018

Contents

1 Introduction **11**
- 1.1 Optimising compilers . . . 11
  - 1.1.1 Dead code elimination . . . 12
- 1.2 Compiler correctness . . . 12
  - 1.2.1 Interactive theorem-provers . . . 12
  - 1.2.2 CompCert . . . 14
- 1.3 CakeML . . . 14
- 1.4 Purpose of this project . . . 15

2 Preparation **17**
- 2.1 HOL4 . . . 17
- 2.2 Starting point . . . 18
  - 2.2.1 Unforeseen limitations of the starting point . . . 19
- 2.3 Project planning and practice . . . 19
  - 2.3.1 Plan of work . . . 19
  - 2.3.2 Third-party tools . . . 21
  - 2.3.3 Backups and version control . . . 21

3 Implementation **23**
- 3.1 Sptrees . . . 23
  - 3.1.1 The next-step function . . . 24
  - 3.1.2 Basic principles of sptrees . . . 24
  - 3.1.3 Implementation and well-formedness . . . 25
  - 3.1.4 Sptrees in this project . . . 26
- 3.2 Code analysis implementation . . . 27
  - 3.2.1 Code analysis in WordLang . . . 28
  - 3.2.2 Code analysis in FlatLang . . . 28
- 3.3 Reachability analysis . . . 30
  - 3.3.1 Specification and reachability relations . . . 30
  - 3.3.2 Reachability implementation . . . 31
  - 3.3.3 Reachability proofs . . . 32

4 Evaluation **37**
- 4.1 Evaluation methods . . . 37
- 4.2 Proof of correctness . . . 37

4.2.1 Proofs of termination 38
4.2.2 Proofs of semantic preservation in WordLang 38
4.2.3 Proofs of semantic preservation in FlatLang 43
4.3 Secondary evaluation 47
4.4 Summary of results 48

5 Conclusion **51**
5.1 Summary of the work completed 51
5.2 Improvements on the work completed 52
5.3 Further work 53

Bibliography **53**

A Intermediate languages **57**
A.1 WordLang 57
A.1.1 Structure of a WordLang program 57
A.1.2 Semantics of WordLang 58
A.2 FlatLang 60
A.2.1 Structure of a FlatLang program 60
A.2.2 Semantics of FlatLang 61

B HOL4 **63**
B.1 Basic principles of HOL 63
B.2 Scripts and theories 64
B.3 The goalstack and interactive proof 65
B.3.1 Tactics 65
B.3.2 Tacticals 66
B.3.3 Conversions and rules 67
B.4 Summary of HOL workflow 67

C Results of code reduction evaluation in FlatLang **69**

D Project Proposal **71**

List of Figures

3.1 A diagram of the key structure of an sptree. 24
3.2 The recursive data type for an sptree. 25
3.3 A diagram of an example sptree. 25
3.4 The implementation of the sptree in Figure 3.3. 26
3.5 A next-step function, represented in sptree form. 27
3.6 The core functions implementing reachability analysis. 31
3.7 The core lemma used in the correctness proof of the reachability functions. 33
3.8 The final result for the proof of correctness of the reachability functions. . 35

4.1 The core lemma used to prove semantic preservation for the optimisation in WordLang. 39
4.2 The final result for the proof of correctness of the optimisation in WordLang. 41
4.3 The core lemma used to prove semantic preservation of the optimisation in FlatLang. 44
4.4 The proof strategy used for verifying the correctness of the optimisation in FlatLang. 46
4.5 The final result of semantic preservation for the optimisation in FlatLang. 47
4.6 Summarised results of code reduction tests for the optimisation in FlatLang. 48

A.1 The data type for a WordLang function. 58
A.2 The record data type for WordLang program state. 59
A.3 The data types for FlatLang operations, expressions, and declarations. . . 61
A.4 The record data types for FlatLang program state and program environment. 62

C.1 The raw results of code reduction tests for the optimisation in FlatLang. . 70

Chapter 1

Introduction

Optimising compilers have had a significant impact in the field of Computer Science since their inception. Previously, compilers had simply translated hand-written assembly code correctly and quickly rather than optimising run time efficiency. However, advances in compiler optimisation allowed powerful high-level languages to be used, safe in the knowledge that these new compilers would transform the code into fast, correct assembly language. This has led to the abundance of high-level languages we see today, and now most languages are commonly used with an optimising compiler – even lower-level languages such as C have powerful optimising compilers such as the GNU Compiler Collection (GCC).

As optimising compilers become more powerful, there is an increasing risk that such a compiler will introduce a fault into critical software prevalent in the modern world. A *verified* optimising compiler is therefore desirable – one that effectively optimises code, but is also proved not to introduce any faults in the resulting binary. Two notable efforts to create a verified compiler are CompCert (§ *1.2.2*) and CakeML (§ *1.3*); the latter is the focus for this project.

This project tackles implementing and verifying global dead code elimination optimisations for the CakeML compiler, in two of its intermediate languages. This chapter contains background information on dead code elimination and optimising compilers (§ *1.1*), compiler correctness (§ *1.2*) with a focus on interactive theorem-provers (§ *1.2.1*), and lastly a brief overview of CakeML (§ *1.3*).

1.1 Optimising compilers

An optimising compiler seeks to maximise some favourable attribute of a compiled executable, or minimise some unfavourable one. For example, a compiler should minimise the time taken to execute a program, and the size of the executable. The general principle is that sacrificing time and resources (such as memory usage) in order to optimise at compile time gives benefits at run time.

Optimising compilers are implemented as several *passes* of transformations: each transforms the program into a semantically-equivalent one, but with some optimisation

applied. The transformations are executed in succession, forming a *pipeline*. All modern compilers have some form of optimisation in their pipelines – for example, GCC allows the user to specify the level of optimisation. The user may want highly optimised binaries, or alternatively a fast compile to speed up an edit-compile-debug cycle of programming.

1.1.1 Dead code elimination

The optimisation considered in this project is dead code elimination, which removes code that does not affect the results of a program (*dead code*). Code which can never be executed due to the control flow of the program (*unreachable code*), or which only affects *dead variables* (variables that are never read), is therefore deleted during compilation. This can reduce the size of compiled binaries and, in the case of dead variable elimination, speed up program execution, as unnecessary instructions are not executed. Dead code elimination can also enable optimisations further down the compilation pipeline, by simplifying program structure.

1.2 Compiler correctness

A *correct* or *verified* compiler does not violate the language specification of any code that it is compiling. Such a compiler is *semantics-preserving*: the semantics of the source code is equivalent to that of the compiled machine code. All observable output of the program is therefore as dictated by the language specification and semantics, and there are no compiler-induced errors (*miscompilations*). In most cases, these guarantees are excessive, but for safety-critical software the potential consequences of miscompilation are not negligible.

There are two main approaches to verifying compiler correctness. *Formal verification* proves semantic preservation through deductive logic (or a similar method), and *compiler validation* tests the compiler rigorously to identify any anomalies. This project focuses on formal methods, using an interactive theorem-prover (§ *1.2.1*).

Compiler verification is not a new concept: the first compiler correctness proof was published in 1967 [9]. Neither is the use of interactive theorem-provers – the same proof was soon automated with Stanford LCF (Logic for Computable Functions – see § *1.2.1*). Compiler verification continues to be a field of research, and the most complete verified compiler to date is CompCert (§ *1.2.2*).

1.2.1 Interactive theorem-provers

Interactive theorem-provers can simplify compiler verification by automating much of the process, allowing verification to take a high-level approach. This is similar to how a correctly-chosen high-level language can simplify the implementation of an algorithm.

The use of formal methods to prove compiler correctness presents significant challenges. A detailed framework describing language semantics is required to specify the semantic properties we wish to preserve, encompassing the source code level down to the assembly code level. This requires a wide range of definitions, from high-level constructs (modules,

objects, types, and more) to machine constructs (instructions, memory, and so on). As a result, verified compilers tend to have many more intermediate languages than non-verified ones: breaking down the compilation into many smaller steps allows the proof of correctness to be broken down into more tractable subproofs.

Transformations between these intermediate languages are complicated, with some optimisation at each stage. This makes it difficult to prove compiler correctness. For example, during compilation functional languages must be transformed to imperative, memory models (such as stack and heap) must be realised, and more besides. For a verified runtime too, garbage collection must be implemented and verified. This will delete data from the heap at run time, which must be shown to have no relevance to the program.

This is too difficult to attempt by hand for a real-world compiler – therefore a theorem-prover is used. *Automated* theorem-provers operate unassisted, searching for a proof automatically – these are not usually appropriate as compiler algorithms are complex, resulting in a large proof search space. The generated proofs can also be indecipherable to humans, making it difficult to update the compiler.

However, *interactive* theorem-provers allow the user to provide the overarching proof strategy, while automating the minutiae of simplification and so on. This reduces the work of the user, but uses their understanding of the compiler algorithms to guide the proof. The generated proofs are a series of guidance steps[1] that make up the user's interaction with the proof system – these are simple and condensed, and so easily reproduced or updated. The degree of automation can vary, from proof checkers, to largely automated systems with minimal guidance.

LCF and successors

A pioneering interactive theorem-prover was *Logic for Computable Functions* (LCF) [4], devised in 1972 by Robert Milner and others as an alternative to the many prior automated-theorem provers[2]. LCF is the predecessor of HOL (the interactive theorem-prover used by CakeML and therefore this project), and has inspired many other interactive theorem-provers, such as Isabelle.

LCF and its successors use a strongly-typed language to define a "theorem" abstract data type – this theorem type can only be constructed by a trusted module implementing primitive inference rules. The strong typing then ensures theorems can be generated only by compounding these inferences – this is an elegant way to guarantee that only valid theorems are created, using well-understood, sound typing systems. The LCF family also has a subgoaling facility: a goal can be broken down into many smaller, more tractable subgoals, which when taken together imply the original. The user simply specifies the method of deconstructing the goal, and the prover generates the subgoals.

[1] In HOL, these are the tactics and tacticals in § *B.3.1* and § *B.3.2*.

[2] Milner said "I was always more interested in amplifying human intelligence than I am in artificial intelligence", referring to his interest in interactive, computer-*assisted* proof as opposed to fully automated proof [6].

The strongly-typed language used in LCF is ML, which was created alongside LCF as a meta-language for creating proof tactics (ML stands for "**M**eta-**L**anguage"). Standard ML is a descendant language of ML, and the ancestor of CakeML.

1.2.2 CompCert

CompCert [7] is the most well-known and complete example of a formally verified optimising compiler. Started in 2005, CompCert compiles a large subset of the C99 programming language (ISO/IEC 9899:1999). The compiler currently targets four architectures, and is specified, implemented, and proved correct using the interactive theorem-prover Coq.

CompCert guarantees that the observable behaviour of compiler-generated code is one of the possible observable behaviours of the input code, as given by the language specification and semantics. The CompCert compilation pipeline involves thirteen transformations to eleven intermediate languages – the proof of semantic preservation is therefore also split into thirteen sub-proofs, one for each pass.

However, CompCert is not verified for all stages of the compilation pipeline, just the "middle-end". A compilation pipeline consists of three stages, the frontend (for parsing, type-checking, and early simplification to give an abstract syntax tree), the middle-end (compilation into an assembly code abstract syntax tree – most optimisations are executed here too), and the back-end (code generation from the abstract syntax tree, assembly, and linking object files into executables). As CompCert is not proven to be semantics-preserving for the front-end and back-end, it is possible for errors to be introduced here.

A fully (or *end-to-end*) verified compiler (verified for all stages of the compilation pipeline) is therefore desirable – CakeML has such a compiler.

1.3 CakeML

The CakeML project [3] is an open-source collaboration started in 2012, aiming to create the CakeML language with an optimising, end-to-end verified compiler. The CakeML language is based on a substantial subset of the functional language Standard ML (SML), which (like the rest of the ML language family) provides static type checking (at compile-time), as well as type inference. High-level abstractions such as higher-order functions, pattern-matching, and polymorphism, make SML useful in both compiler-writing and theorem-provers[3].

The interactive theorem-prover HOL4 (§ *2.1*) is used to specify the language semantics and compiler in higher-order logic, The core of CakeML is a translator for functions specified in HOL to CakeML functions [10]. This generates a proof for each translation, stating that the input HOL function is semantically equivalent to the resulting CakeML function. The CakeML compiler is written in HOL, then translated to CakeML. Evaluating the HOL compiler specification on the CakeML compiler implementation (the output of the translator) produces a verified implementation of the compiler – this is similar to the standard approach to bootstrapping a compiler [8]. The front-end and middle-end of the

[3]HOL4, the theorem-prover used with CakeML, is a library for theorem-proving in SML.

compiler are specified in HOL and verified in this way. Validated models of instruction set architectures allow verified translations to architecture-specific instruction sets too, and so CakeML achieves an end-to-end verified compiler. This sets it above CompCert, which only verifies the middle-end. So far, verified binaries have been produced for the x86-64, ARM, RISC-V and MIPS architectures.

CakeML is continually under development, and its current version passes through twelve intermediate languages. The subset of SML supported has grown dramatically since the start of the project, and many optimisations have been successfully implemented and verified too.

1.4 Purpose of this project

This project implements an inter-procedural (global) dead code elimination optimisation for two intermediate languages of the verified, optimising compiler CakeML. The optimisation is proved to preserve semantics in both languages using the interactive theorem-prover HOL4.

Currently, even simple CakeML programs take a long time to compile, as the large "basis" library (mirroring the SML basis library) is prepended onto any code before compilation. A lot of unnecessary code is therefore compiled – a global dead code elimination pass would ideally remove this early on in the compilation pipeline. This pass was implemented and verified for two separate languages – one early in the pipeline, the other late.

Chapter 2

Preparation

This chapter details the preparation work undertaken for this project prior to most of the implementation.

The main preparation for this project was familiarisation with the interactive theorem-prover HOL4 (referred to as "HOL" in this document). The CakeML language specification and compiler algorithm are written in HOL, and all compiler proofs are created within the HOL system. Familiarity with HOL is therefore necessary to develop on the CakeML codebase, but this was no small task: the HOL documentation estimates that one full month is required to have a basic understanding of HOL, when starting from scratch. This is because proving in HOL is very different to coding in imperative/functional languages, and requires a distinct skill set.

The project involves two of the CakeML intermediate languages (WordLang and FlatLang), so familiarity with these was required too. An overview of these languages can be found in *Appendix A*.

This chapter therefore gives an overview of the HOL system (§ *2.1*), specifies the starting point of the project (§ *2.2*), and details the plan of work (§ *2.3*).

2.1 HOL4

Familiarisation with HOL was achieved through careful examination of the HOL documentation, and supervisions with Anthony Fox. This process continued throughout the project – in particular many concepts were elucidated by the first stages of proof on the optimisation itself.

A brief introduction to HOL follows – more information on the basic tools used for this project can be found in *Appendix B*, and in the HOL4 documentation [1] [2].

HOL (which stands for "**H**igher-**O**rder **L**ogic") is a tool for aiding proofs in higher-order logic using sequents. It provides a programming environment for proving theorems and constructing other proof tools, as well as many decision procedures and automated provers which can discharge simple proofs – this allows the user to focus on the higher-level

proof strategy. HOL is a widely-used proof tool at the cutting edge of research[1].

HOL is implemented as a library in SML, and effectively provides an abstraction layer above SML for interaction with higher-order logic. Theorem-proving is essentially programming in SML, with extra functions and handles for manipulating higher-order logic constructs such as terms and theorems.

Interaction through plugins for the Vim and Emacs editors to manage a *goalstack* of active desired proofs (§ *B.3*), and powerful proof tools known as *tactics* (§ *B.3.1*) and *tacticals* (§ *B.3.2*) make HOL a powerful aid in theorem-proving, but also a difficult one to master. HOL also allows for the compilation of scripts, which contain definitions, axioms, and theorems (with their proofs), into *theories* (§ *B.2*). These are re-usable, modular units – for example the HOL codebase contains `boolTheory`, encapsulating Boolean theory.

Familiarisation with the workflow of interactive specification, proof, and theory export (§ *B.4*) using the Vim editor was achieved with the help of Anthony Fox and Magnus Myreen. Without supervision from experienced mentors, learning to use HOL is a difficult task, in part due to the many available tools and limited documentation.

2.2 Starting point

This section details the starting point of this project. Note in particular the limitations of the starting point for the FlatLang intermediate language (§ *2.2.1*) which complicated work on the optimisation for FlatLang.

The CakeML codebase is open-source, and can be found on GitHub[2]. It includes the specification and semantics of the language, and the compiler algorithm, all defined and verified in higher-order logic using the HOL4 interactive theorem-prover. The compiler is proven to be semantics-preserving, and many compiler optimisations have been implemented and verified.

This project focuses on the intermediate languages FlatLang and WordLang, the first and the antepenultimate intermediate languages respectively. The specifications and semantics of both of these languages are part of the CakeML codebase, as are the algorithms for compilation to/from them. The optimisations written in this project will be added as transformations within the languages themselves. Each language has a number of optimisations already written and verified – these proofs provided some inspiration for the project implementation, and some useful theorems. Many results in the CakeML codebase were pre-existing, but any results about the new optimisations were written as part of the project – however, a number of general results about the CakeML codebase also had to be proved. There are GitHub issues that generated this project[3].

[1] In 2017, a course on HOL was given at KTH, Stockholm for advanced master students, consisting of approximately thirteen hours of study each week for nine weeks [11].

[2] `https://github.com/CakeML/cakeml`.

[3] `https://github.com/CakeML/cakeml/issues/336` and `https://github.com/CakeML/cakeml/issues/337`.

The HOL4 interactive theorem-prover is also open-source, and can also be found on GitHub[4]. It has tutorials, documentation, and guidebooks available (though these are incomplete), as well as a reference library. The implementation of sptrees (a key data structure in this project – see § *3.1*) is part of the HOL libraries, and it comes with many useful theorems. Many further definitions and theorems were created as part of this project, and some of these were incorporated into the latest version of HOL. I had no prior experience with HOL or any other form of computer-assisted proof before starting this project.

2.2.1 Unforeseen limitations of the starting point

The FlatLang intermediate language did not exist on the `master` branch of the CakeML GitHub repository at the time of writing, as it has not yet superseded its predecessor, ModLang. This is an issue with the CakeML development timeline rather than this project: at the beginning of the project, FlatLang was expected to become part of the `master` branch in January 2018. However, this has not happened at the time of writing (May 18, 2018). Therefore one of the success criteria cannot be achieved: the optimisation in FlatLang cannot be integrated into the latest version of the CakeML compiler.

Work on both dead code elimination implementations therefore primarily used the `type+module-update` branch of the repository, which contains the latest version of FlatLang. Both optimisations have been integrated into this branch, which therefore achieves the integration success criterion to the fullest extent possible.

The success criterion of demonstrating the effect of the optimisation in FlatLang on code size was not possible until late on in the project: the basis library on the `type+module-update` branch did not build until May 7, 2018[5], preventing the compilation of any code. The extension aim of empirically determining the efficiency of the optimisations is not possible as the remainder of the compiler does not build, preventing any benchmarking.

2.3 Project planning and practice

This section details the planning of the project and the key practices followed over its course. In particular, the plan of work is detailed, followed by an overview of the third-party tools used, and the backup and version control practices employed.

2.3.1 Plan of work

This project is well-suited to the waterfall method: it adds to a larger project and so has a well-understood progression. It can be further split into three sub-projects, and each of these was implemented using an evolutionary model: the core implementation was completed, and then gradually refined while attempting to prove its correctness. The last two sub-projects were executed in parallel, due to the similarities between them.

[4]`https://github.com/HOL-Theorem-Prover/HOL`.

[5]Commit URL: `https://github.com/CakeML/cakeml/commit/52ebb2`.

The plan for the project is detailed below:

1. **Requirements and preparation**: familiarise with the HOL system and the sptree data structure, with supervision from Anthony Fox.
2. **Sub-project (core aim)**: abstract reachability functions.
   (a) *Implementation*: implement the abstract reachability functions operating on sptrees, with supervision from Magnus Myreen.
   (b) *Verification*: prove correctness of these functions, with feedback into the implementation allowing refinement. Supervised by Anthony and Magnus.
3. **Sub-projects (core and extension aims)**: optimisations in WordLang and in FlatLang.
   (a) *Requirements and preparation*: familiarise with the FlatLang and WordLang languages, with supervision from Magnus.
   (b) *Implementation (core aim)*: implement the FlatLang and WordLang optimisations roughly in parallel.
   (c) *Verification (core and extension aims)*: prove the correctness of both optimisations roughly in parallel, with supervision from Magnus and feedback into the implementation (as before). The WordLang verification is a core aim, while the FlatLang verification is an extension aim.
4. **Evaluation (core and extension aims)**: run tests from the CakeML codebase, demonstrate the benefits of the optimisations, and identify areas for improvement in future work.

This plan differs to the originally proposed plan of work: the optimisation in WordLang was moved to a much later stage of the project, as the WordLang and FlatLang implementations and proofs have features in common. Executing them in parallel is therefore sensible to avoid repeated work. The implementation of the abstract reachability functions was therefore the first task – this also is sensible, as it does not involve the CakeML codebase and so can be achieved before familiarisation with CakeML is complete. It also allows more experience with HOL before tackling CakeML proofs.

There were some unexpected changes to the timeline of the project: the verification of the WordLang optimisation took significantly longer than planned. This optimisation in WordLang was intended as a starter project, but due to the low-level nature of WordLang, its proofs were more involved than expected (§ *4.2.2*). Evaluation was also not possible until late on in the project (§ *4.3*).

Disruptions to the plan of work

The evaluation stage of the project was impeded due to the delay in the CakeML development timeline, detailed in § *2.2.1*. As a result, the optimisation in FlatLang could not be integrated into the `master` branch, regression tested, nor benchmarked.

2.3.2 Third-party tools

The two third-party tools used in this project are the CakeML codebase, and the HOL4 interactive theorem-prover. Both are open-source, and are described in § *2.2*.

2.3.3 Backups and version control

The project was carried out entirely on my own personal computer, a Dell XPS L702X, with an Intel i7-2720QM processor and 8 GB of RAM. It is dual-booted with Windows 10 and Ubuntu 16.04 LTS – working with HOL was far simpler using Ubuntu. Frequent backups were made to Google Drive and to a personal flash drive. The failure of my personal computer would therefore not affect project progress.

Git was used as a version control system, with the open-source CakeML repository hosted on GitHub. Project work used either local repositories or the GitHub-hosted repository, and was later integrated into the `type+module-update` branch of the GitHub repository.

Chapter 3

Implementation

This project falls into two distinct parts: implementing the optimisation, and verifying its correctness. During the project, these were achieved in parallel – the implementation inspires the proof of correctness, and difficulties in proving correctness necessitate changes to the implementation. However for this document, the implementation more naturally falls into this chapter, and the validation mostly into the next chapter.

The implementation of this project falls into three further sub-tasks, each of which is examined in turn:

1. A next-step function is computed. This function is represented using an sptree associative data structure (§ *3.1*), and maps each global variable (for FlatLang) or function number (for WordLang) to a set of global variables/function numbers on which it depends. The function is computed by analysing the FlatLang or WordLang intermediate code (§ *3.2*).

2. This next-step function is used to determine which global variables/functions are reachable (according to a specification of reachability) from a set of starting ones (§ *3.3*), and unreachable ones are removed.

3. These stages are stitched together to form the standalone optimisation (§ *3.2*) – this modular transformation is then simply added as another stage to the CakeML compilation pipeline.

This implementation can be found in the `type+module-update` branch of the CakeML GitHub repository, which contains the most up-to-date version of FlatLang[1].

3.1 Sptrees

Simplified Patricia trees ("sptrees") are a key data data structure in this project, and are implemented as part of the HOL libraries. They will be used to represent the next-step function.

[1]As discussed in § *2.2.1*, FlatLang has not superseded its predecessor, ModLang, on the `master` branch at the time of writing.

3.1.1 The next-step function

An associative data structure is needed to represent the next-step function: each key represents a global variable (for FlatLang) or function number (for WordLang), and the corresponding value represents a set of global variables/function numbers reachable from the key in a single step. Both global variables and function numbers are represented as natural numbers in the CakeML compiler – the structure therefore maps natural number keys to sets of natural numbers.

Sptrees are such a data structure, and are optimised for use with HOL.

3.1.2 Basic principles of sptrees

Sptrees are tree structures in which any node in the tree is either empty or stores a value. Each node has up to two children, so lookup of a node to become an efficient bit-by-bit key comparison to branch left or right at each stage, giving $\mathcal{O}(n)$ lookup for an n-bit key. Sptrees have natural numbers as keys, and are specialised for use in HOL – in HOL, natural numbers are implemented as numerals:

$$N ::= \textbf{ZERO} \mid \textbf{BIT2}\ N \mid \textbf{BIT1}\ N\ ,$$

where **ZERO** represents zero, (**BIT2** n) represents $(2n + 2)$, and (**BIT1** n) represents $(2n + 1)$.

The structure of an sptree mirrors this: the root node is keyed by zero, and the left and right subtrees are keyed by $n \mapsto (2n + 2)$ and $n \mapsto (2n + 1)$ mappings of the root tree respectively (*Figure 3.1*). Lookup of a node therefore branches left for an even key and right for an odd key, recursively stripping down a single **BIT2** or **BIT1** constructor at each branch.

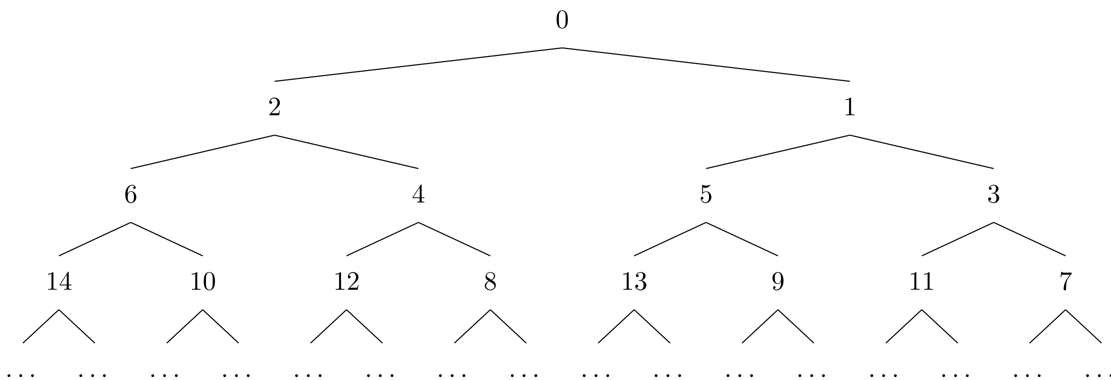


Figure 3.1: *A diagram of the key structure of an sptree. Note that the subtree rooted at node 1 is a mapping of the tree rooted at node 0, in which every key n is mapped to $(2n + 1)$. Similarly the subtree rooted at node 2 is a mapping $n \mapsto (2n + 2)$.*

3.1.3 Implementation and well-formedness

```
1 datatype 'a sptree =
2     LN                                              (* Leaf-None *)
3   | BN of 'a sptree * 'a sptree                     (* Branch-None *)
4   | LS of 'a                                        (* Leaf-Some *)
5   | BS of 'a sptree * 'a * 'a sptree;               (* Branch-Some *)
```

Figure 3.2: *The recursive data type for an sptree containing values of type α. Values can be stored at any* ***LS*** *or* ***BS*** *node in the tree.*

Figure 3.2 shows the implementation of the recursive data type for an sptree, written in Standard ML. In HOL, the constructors can be curried: `(BN LN LN)` can be used instead of `BN(LN,LN)`. The HOL curried constructors will be used from now on. The four constructors represent the following:

- **Leaf-None, `LN`.** A terminal node (at the bottom of a tree), containing no key-value pair.
- **Branch-None, `BN t1 t2`.** A binary branching node (within the tree), containing no key-value pair, and branching to two subtrees, `t1` and `t2`.
- **Leaf-Some, `LS a`.** A terminal node with a single key-value pair, with value `a`.
- **Branch-Some, `BS t1 a t2`.** A branching node with a single key-value pair, with value `a` and subtrees `t1` and `t2`.

The sptree in *Figure 3.3* can therefore be represented by the code in *Figure 3.4*.

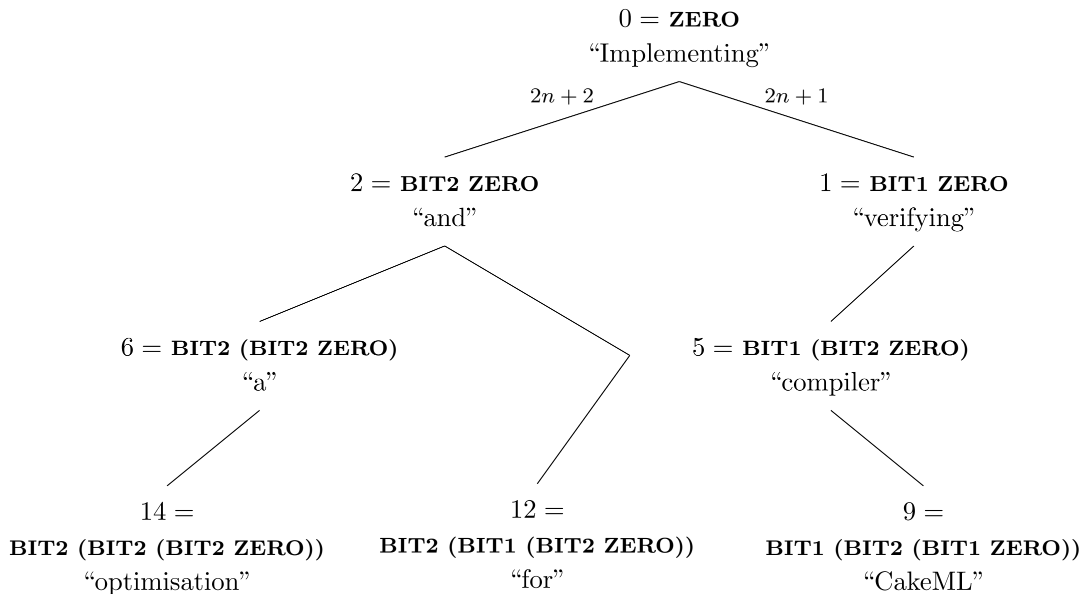


Figure 3.3: *A diagram of an sptree, containing* ***string*** *values. This sptree contains the following key-value mappings: {0 ↦ "Implementing", 1 ↦ "verifying", 2 ↦ "and", 5 ↦ "compiler", 6 ↦ "a", 9 ↦ "CakeML", 12 ↦ "for", 14 ↦ "optimisation"}.*

```
BS
    (BS
        (BS (LS "optimisation") "a" LN)
        "and"
        (BN (LS "for") LN)
    )
    "Implementing"
    (BS
        (BS LN "compiler" (LS "CakeML"))
        "verifying"
        LN
    )
```

Figure 3.4: *The implementation of the* **`string`** *sptree in* Figure 3.3.

Sptree sets

Sptrees are useful to represent sets of natural numbers: a structure of type `unit sptree` can store a `unit` value at every number in the set.

An sptree set should be unique: there should be only one sptree representation of any set. HOL defines a well-formedness property (`WF` *tree*) ensuring that sptrees do not contain redundancy: for example, trees of the form `(BN LN LN)` and `(BS LN a LN)` could be simplified to `LN` and `(LS a)` respectively. A well-formed tree has no such redundancy, and HOL provides a function which converts trees to well-formed equivalents storing the same key-value pairs. Under this property, each set of keys has a unique representation.

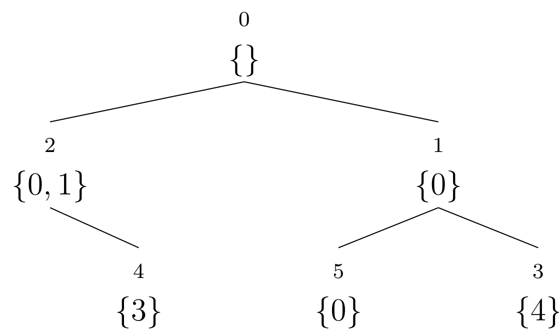
3.1.4 Sptrees in this project

A structure of type `((unit sptree) sptree)` is used to represent a next-step function: the natural number keys correspond to global variables/function numbers, mapping to sptree sets of variables/function numbers.

For example, the pseudocode in *Figure 3.5a* can be represented as the sptree next-step function in *Figure 3.5b*. If the global variable `global_2` is the entry point of this program, then clearly global variables `global_3`, `global_4`, and `global_5` have no effect on the program – these should be removed by a dead code elimination pass.

```
val global_0 := 0;
val global_1 := lookup global_0 + 1;
val global_2 := (* entry point *)
        lookup global_1 * lookup global_0;
val global_3 := lookup global_4;
val global_4 := lookup global_3;
val global_5 := lookup global_0;
```

(a) *Pseudocode for a program consisting of a series of global assignment and lookup operations. The* `lookup` *operation is effectively a dereference of a global variable, and* `global_2` *is the entry point of the program. This pseudocode can be considered an oversimplification of a simple* FlatLang *program –* FlatLang *programs consist of a series of top-level declarations (*§ A.2.1*).*



(b) *The next-step function of the pseudocode in* Figure 3.5a *in the form of an sptree, mapping each global variable to the set of global variables on which it depends. The value stored in* `global_2` *depends on* `global_0` *and* `global_1`*, hence the value for key* 2 *in the data structure is the set* $\{0, 1\}$.

Figure 3.5: *An example of a* `unit sptree sptree` *data structure used to represent a next-step function for global variables.*

3.2 Code analysis implementation

The first step in any compiler optimisation is an analysis of the program, to determine what code transformation (if any) is *safe*, in that it cannot affect the observable behaviour of the program.

This section details these code analyses implemented for both WordLang (§ *3.2.1*) and FlatLang (§ *3.2.2*), and the corresponding code removal functions. The implementation in FlatLang is the most useful for CakeML development: the compiler currently prepends the entire basis library onto any user-written code, so a global dead code elimination pass early on in the compilation pipeline would remove most of this and reduce work for later stages. The optimisation in WordLang was intended as a simple introduction to the implementation and its proof, simplifying the FlatLang case – however, the WordLang

proof of correctness was more difficult than anticipated (§ *4.2.2*).

3.2.1 Code analysis in WordLang

This section describes the code analysis for WordLang, which returns a next-step function (§ *3.1.1*) and later executes the code removal.

A WordLang program (§ *A.1.1*) consists of a list of *3*-tuples, each encapsulating a low-level, assembly-like function: the first item of each tuple is a WordLang function number in the domain of the next-step function. It maps to a set given by analysing the third item in the tuple, a WordLang `prog` term which encapsulates the corresponding WordLang function body. The `prog` term is recursively searched for `Call` or `LocValue` expressions, which can reference other function numbers. For the `Call` case, the target of the call is in the next-step set, as are any function numbers found in the return and exception handler code. For the `LocValue` case, the location referenced is in the set. For all other expressions, the next-step set is empty.

The next-step function is accumulated by analysing each function in the list in this way, and combining their results – the function `analyseWordCode` implements this.

Overall, the optimisation is implemented in the function `removeWordProg`, which executes the following:

1. Take in the program entry point and list of functions as arguments. Note that for WordLang, the optimisation cannot determine the program entry point itself.

2. Analyse the list of functions (using the `analyseWordCode` function) to give a next-step function. Ensure this satisfies the `WF_SET_TREE` predicate with a call to `mk_wf_set_tree` (§ *3.3.3*, *Equation 3.2*).

3. From the next-step function and program entry point, determine a set of all reachable WordLang functions using the abstract reachability analysis function `closure_spt` (§ *3.3*).

4. Filter the list of WordLang functions to remove any that are not determined to be reachable.

3.2.2 Code analysis in FlatLang

This section describes the code analysis functions which determine a next-step function from a FlatLang program, and the functions which later execute code removal.

FlatLang programs and expressions are more complicated than WordLang ones (§ *A.2.1*) – this complicates the analysis pass too. A FlatLang program is a list of top-level declarations of values, types, and exceptions, which are sequentially executed. The value declarations contain expressions, which can contain operations. The analysis determines which global variables (initialised by operation `(GlobalVarInit : Op)`) may lookup which others (referenced by operation `(GlobalVarLookup : Op)`). Pseudocode for a highly simplified FlatLang program is shown in *Figure 3.5a*.

Building the next-step function

The list of declarations is processed item-by-item into a next-step function and starting set of immediately reachable global variables by the function `analyseCode`. Each expression in a value declaration (of the form `Dlet exp`) is analysed to form a starting set and some mappings in the next-step function. The latter is easily achieved by mapping all possible global variable initialisations to all possible global variable lookups in the expression, to ensure safety.

Determining the starting set is more subtle. Some declarations have observable effects, even if their global variables are not reachable in the sense of the optimisation (for example, a `Raise` expression observably causes an exception irrespective of global variables) – these should not be removed.

Declarations are therefore categorised on their purity and whether they are “hidden”.

Pure expressions. For this analysis, these have no immediate effect on program state other than assigning to global variables: assigning to a global variable that is never read is unobservable behaviour, but all other effects on state may be observable. Pure expressions are therefore candidates for removal – conversely, all impure expressions must have all global variable initialisations and lookups in the starting set.

Hidden expressions. These are not executed unless their global variables are referenced (for example, a function body is not executed unless the function is called via its global variable). Their global variables are therefore not added to the starting set, and only mappings in the next-step function are considered. However, all other functions must be assumed to be immediately reachable.

This is a coarse-grained strategy, implemented when difficulties were encountered while attempting to prove the correctness of the initial optimisation implementation. This is an example of the trade-off between efficacy of the optimisation and ease of proof of correctness.

Note that the FlatLang analysis determines the starting set itself, a key difference to the WordLang case which has the program entry point supplied to it.

The overall optimisation

Overall the optimisation is implemented in the function `removeFlatProg`, which executes the following:

1. Take in the list of declarations as an argument.
2. Analyse the list of declarations (using the `analyseCode` function) to give a next-step function, and a starting set of immediately reachable global variables. Ensure the next-step function satisfies the `WF_SET_TREE` predicate using `mk_wf_set_tree` (§ *3.3.3*, *Equation 3.2*).

3. From the next-step function and starting set, determine a set of all reachable global variables using the abstract reachability analysis function `closure_spt` (§ *3.3*).

4. Filter the list of declarations to remove any that are both pure and do not initialise any reachable global variables. These clearly cannot have any observable effects.

3.3 Reachability analysis

This was the first part of the project to be implemented: conceptually it is relatively simple, so the proof of its correctness was the main focus.

The reachability analysis takes in a starting set of reachable global variables (for FlatLang) or function numbers (for WordLang), and a next-step function over global variables/function numbers created by the code analysis pass (§ *3.2.1* for WordLang, and § *3.2.2* for FlatLang). It returns the set of all nodes reachable in the next-step function from the starting set, which might therefore be reached during program execution and so should not be removed. The specification for this analysis (§ *3.3.1*) and the core implementation (§ *3.3.2*) follow, as well as the proof that the implementation meets the specification (§ *3.3.3*).

3.3.1 Specification and reachability relations

This reachability analysis must follow some specification of reachability to be useful to the optimisation. This is specified using the abstract reachability relations (*Equation 3.1*). All variables are italicised, and all HOL functions are monospaced – this convention will be used for the remainder of this document.

$$\begin{aligned}\texttt{isAdjacent}\ \mathit{tree}\ x\ y \overset{\text{def}}{\Leftrightarrow} \exists\ \mathit{setA}\ \mathit{setB}.&\\ &(\texttt{lookup}\ x\ \mathit{tree} = \texttt{SOME}\ \mathit{setA})\ \wedge (y \in \texttt{domain}\ \mathit{setA}) \wedge\\ &(\texttt{lookup}\ y\ \mathit{tree} = \texttt{SOME}\ \mathit{setB})\ ,\\ \texttt{isReachable}\ \mathit{tree} \overset{\text{def}}{=} (\texttt{isAdjacent}\ \mathit{tree})^{*}\ .&\end{aligned} \tag{3.1}$$

where the `isReachable` relation is the reflexive, transitive closure (RTC) of the `isAdjacent` relation. Both take a next-step function parameter (*tree*), and relate two nodes x and y. The `isAdjacent` relation states that node y can be reached in a single step from node x (they are adjacent) if lookup of x gives next-step set *setA*, which contains y. Furthermore, y is in the domain of the next-step function (and so can be looked up to give *setB*). The `isReachable` relation (as the RTC of this) relates any node to itself, and to all nodes reachable in any number of `isAdjacent` steps. RTC induction and rewrite theorems are usefully provided by the HOL system as part of `relationTheory`.

The reachability analysis therefore takes in a starting set of nodes and a next-step function, and returns the sptree set of all nodes that are reachable in the next-step function (according to `isReachable`) from any of the starting nodes.

3.3.2 Reachability implementation

```
fun close_spt reachable seen tree =
    let val toLook = (difference seen reachable) in
        if toLook = LN then reachable else
        let val index = (getOne toLook) in
            case (lookup index tree) of
                NONE => reachable
              | SOME new => close_spt (insert index () reachable)
                              (union new seen) (delete index tree)
        end
    end;

fun closure_spt start tree = close_spt LN start tree;
```

Figure 3.6: *The core functions implementing reachability analysis. Arguments* `reachable` *and* `seen` *are sptree representations of sets, and so the* `difference`*,* `delete`*, and* `union` *functions are analogous to the set operations of the same name. The* `insert` *operation on line 7 adds the value* `index` *to the set* `reachable`*, and the function* `getOne` *returns a value from a non-empty set. The last argument* `(tree : unit sptree sptree)` *represents a next-step-function (§ 3.1.4) –* `(lookup index tree)` *in line 5 therefore returns an sptree set of all nodes reachable from* `index` *in a single step. The separation of these two functions (even though* `close_spt` *is simply an auxiliary function for* `closure_spt`*), enables proofs over the properties of* `close_spt` *independently of* `closure_spt` *(§ 3.3.3).*

The `closure_spt` function (line 12) is a graph search of the next-step function `tree` from a set of root nodes `start`. This follows a similar pattern to the tricolour algorithm [5], used for tracing live references for heap garbage collection. This algorithm maintains three separate sets of nodes: a *white* set of unencountered nodes, a *black* set of fully explored nodes (all outgoing edges have been traversed), and a *grey* set of encountered but not explored nodes. The algorithm iteratively takes a grey node and explores all of its outgoing edges: encountered nodes are coloured grey, and the parent node is coloured black once its edges have been examined. When there are no more grey nodes, the algorithm terminates: all black nodes are then reachable, all white ones unreachable. This must hold, as black nodes cannot have edges to white ones (or else the white nodes would have been turned grey).

In the `close_spt` function, `(reachable : unit sptree)` is the set of black nodes, `(toLook : unit sptree)` is the set of grey nodes (all other nodes are white). On each recursive call, a node `index` is taken from the grey `toLook` set (line 4) and its outgoing edges are traced by lookup of `index` in `tree`. Encountered `new` nodes are added to the `seen` set (line 8) from which `toLook` is computed, and `index` is moved to the black `reachable` set.

There are two termination cases for `close_spt`: the standard case when the grey set is empty (line 3), and one due to the sptree representation of the next-step function. As each node is explored, it must be looked up in `tree` to find a next-step set. If the node is not in the domain of the next-step function, the algorithm fails due to malformed input in the `seen` argument (line 6). However this case should not occur for sensible inputs, which is important for the proof of correctness for this function: a contradiction in the assumptions will be derived to show that this case never occurs (§ *3.3.3*).

There are some key differences with the tricolour algorithm. A grey set is not explicitly maintained – instead a set of `seen` nodes represents all nodes seen so far (the union of the grey and black sets). The grey set `toLook` is then computed as the difference between the `seen` set and the black set (`reachable`) – this prevents re-exploration of nodes encountered for the second time. This implementation is also destructive on its input: it deletes nodes from `tree` on each recursive call. This simplifies the proof that this function terminates (§ *4.2.1*).

3.3.3 Reachability proofs

The reachability functions are only useful if shown to obey their specification (§ *3.3.1*). Note that the proofs in this subsection do not involve any of the CakeML codebase, and so were a useful introduction to proof in HOL.

A more general lemma (*Figure 3.7*) about the helper function `close_spt` had to be proved in order to derive the desired results for the more specialised `closure_spt` function (*Figure 3.6*). The helper function is more general, so more powerful function invariants can be used as assumptions in the lemma. Once this lemma is proved, it can be specialised into the desired result (*Figure 3.8*) and many assumptions will then trivially hold. Attempting to prove the desired theorem directly results in a dead end, due to the lack of sufficiently powerful assumptions concerning the function arguments.

Lemma for `close_spt`

This lemma was arrived at by careful reasoning about `close_spt` (*Figure 3.6*) to determine some of its invariants. It was developed iteratively: attempts to prove initial formulations of the lemma resulted in dead ends, so new assumptions were introduced to enable the proof. The desired lemma is stated in *Figure 3.7*.

The lemma examines an arbitrary recursive call in the execution of `close_spt`, and the assumptions are invariants which hold between recursive calls. The terms *fullTree* and *roots* represent the initial next-step function and starting set of nodes (respectively) in the initial call to `closure_spt` – note that they are never explicitly stated as such, but have the same key properties (expressed in the assumptions). These terms are the essence of those initial arguments, encapsulating enough of their properties to give the correct results (even if the terms may never arise in any execution pattern). In particular *roots* is a mathematical set rather than an sptree set.

This lemma should be examined in more detail:

$$\begin{array}{c}
1.\ \texttt{WF}\ \mathit{reachable} \qquad 2.\ \texttt{WF}\ \mathit{seen} \qquad 3.\ \texttt{WF}\ \mathit{tree} \\
4.\ \texttt{WF_SET_TREE}\ \mathit{fullTree} \qquad 5.\ \texttt{subspt}\ \mathit{reachable}\ \mathit{seen} \qquad 6.\ \texttt{subspt}\ \mathit{tree}\ \mathit{fullTree} \\
7.\ \forall\ n.\ n \notin \texttt{domain}\ \mathit{reachable} \Rightarrow \texttt{lookup}\ n\ \mathit{tree} = \texttt{lookup}\ n\ \mathit{fullTree} \\
8.\ \forall\ k.\ k \in \texttt{domain}\ \mathit{seen} \Rightarrow \exists\ n.\ (n \in \mathit{roots} \land \texttt{isReachable}\ \mathit{fullTree}\ n\ k) \\
9.\ \forall\ k.\ k \in \texttt{domain}\ \mathit{reachable} \Rightarrow (\forall\ a.\ \texttt{isAdjacent}\ \mathit{fullTree}\ k\ a \Rightarrow a \in \texttt{domain}\ \mathit{seen}) \\
10.\ \mathit{roots} \subseteq \texttt{domain}\ \mathit{seen} \qquad 11.\ \mathit{roots} \subseteq \texttt{domain}\ \mathit{fullTree} \\
\hline
\texttt{domain}\ (\texttt{close_spt}\ \mathit{reachable}\ \mathit{seen}\ \mathit{tree}) = \\
\{a \mid \exists\ n.\ \texttt{isReachable}\ \mathit{fullTree}\ n\ a \land n \in \mathit{roots}\}
\end{array}$$

Figure 3.7: *The desired lemma for the correctness proof of the abstract reachability functions. As before, variables are italicised and HOL functions are monospaced.*

Assumptions. Every assumption must be an invariant preserved between recursive calls, and must be a tautology when this lemma is specialised later for `closure_spt`.

- **1, 2, 3, and 4.** The various sptrees in this algorithm should be well-formed (indicated by the predicate `WF`). This invariant holds because the various sptree operations used (`difference`, `insert`, `union`, and `delete`) all preserve well-formedness[2], and is necessary as `getOne` may fail on sptrees that are not well-formed.

  The fourth assumption can be expanded:

  $$\texttt{WF_SET_TREE}\ \mathit{fullTree} \overset{\text{def}}{\Leftrightarrow} (\forall\ x\ y.\ \texttt{lookup}\ x\ \mathit{fullTree} = \texttt{SOME}\ y \Rightarrow \texttt{domain}\ y \subseteq \texttt{domain}\ \mathit{fullTree}\ \land\ \texttt{WF}\ \mathit{fullTree}). \tag{3.2}$$

  This ensures that any set in the range of the next-step function is a well-formed (and so unique) sptree set, and each value in the set is also in the domain of the next-step function: there are no spurious nodes in the range of the next-step function that cannot themselves be looked up.

  An accompanying function (`mk_wf_set_tree`, also written as part of the project) transforms any sptree next-step function into one with the same stored values which satisfies this predicate. It may be possible to dispense with this predicate, and derive the result from the language semantics (§ *5.3*).

- **5 and 6.** The set of *reachable* nodes must be a subset of the ones *seen* so far, and the *tree* next-step function must be a subgraph of the initial next-step function, *fullTree* (recall a node is deleted from the next-step function on each recursive call). This is expressed with the `subspt` relation, representing sub-sptrees: a sub-sptree has a subset of the keys of its super-sptree, and has the same value stored at each key as the super-sptree.

[2]These proofs are in `sptreeTheory`, except for the proof for `difference` – this was one of the many sptree lemmas proved in this project.

- **7.** Nodes are deleted from the next-step function when they are added to the *reachable* set, so any nodes in the modified next-step function *tree* but not in the *reachable* set must also be in the original next-step function *fullTree*.
- **8.** Any nodes in the *seen* set must be reachable, otherwise they would not have been encountered.
- **9.** The next-step nodes (according to `isAdjacent`) of any node in the *reachable* set must have been encountered. These are the grey nodes of the tricolour algorithm.
- **10 and 11.** The initial starting set (*roots*) must be a subset of both the nodes *seen* so far and the domain of the initial next-step function *fullTree*.

Conclusion. At any recursive call, (`close_spt` *reachable seen tree*) should return an sptree set of numbers which are reachable in the initial next-step function *fullTree* from any node in the starting set *roots*. This relates the abstract reachability relation `isReachable` to the abstract reachability function `close_spt` in the desired manner.

The proof strategy for this lemma is an induction over the series of recursive calls to `close_spt`. The induction theorem automatically generated by HOL when the function was defined is used to start the proof. Overall, the proof shows that if a step in the recursion satisfies the desired properties, then so does the previous one.

Each termination case of the function is proved to satisfy the desired properties:

Case (*toLook* = `difference` *seen reachable* = `LN`)**.** This is the expected termination case, when the search for reachable nodes has been exhausted. The sets *seen* and *reachable* are therefore equal. By the ninth assumption, any node in the *reachable* set has its adjacent nodes also in the *reachable* set (as it is equivalent to the *seen* set). The *reachable* set is therefore closed under `isAdjacent`. As the *roots* set is a subset of the *seen* set by the tenth assumption, and `isReachable` is the RTC of `isAdjacent`, the returned value (*reachable*) must satisfy the conclusion.

Case (`lookup` *index tree* = `lookup` (`getOne` *toLook*) *tree* = `NONE`)**.** This can only occur on malformed input, and so is proved by deriving a contradiction in the assumptions. The *index* node is not in the domain of the next-step function *tree*, and (*toLook* = `difference` *seen reachable*). Therefore *index* is in the *seen* set but not in the *reachable* set. By the eighth assumption, *index* is reachable from the *roots* set in *fullTree*. However to be reachable, either *index* is in the domain of *fullTree*, or is in *roots* (and so in the domain of *fullTree* by the eleventh assumption). But by the seventh assumption (`lookup` *index tree* = `NONE` = `lookup` *index fullTree*), so *index* is not in the domain of *fullTree*, giving a contradiction.

Recursive case. This case requires instantiation of the inductive hypothesis, which states that if all assumptions are satisfied for the recursive case, then the recursive call

satisfies the initial goal. The goal therefore becomes:

$$\texttt{domain}\\ (\texttt{close_spt}\ (\texttt{insert}\ \mathit{index}\ ()\ \mathit{reachable})\ (\texttt{union}\ \mathit{new}\ \mathit{seen})\ (\texttt{delete}\ \mathit{index}\ \mathit{tree})) = \\ \{a \mid \exists\, n.\ \texttt{isReachable}\ \mathit{fullTree}\ n\ a \wedge n \in \mathit{roots}\}\ . \tag{3.3}$$

Provided the assumptions in *Figure 3.7* hold for the new arguments (`insert` *index* () *reachable*), (`union` *new seen*), and (`delete` *index tree*) (as they did for original arguments *reachable*, *seen*, and *tree*) then the goal is a direct result of the inductive hypothesis. These can be shown to hold due to the properties of the sptree operations used, and the derivations of the terms *new* and *index* – the details are omitted here.

The conceptual simplicity of this proof made it a useful introduction to HOL. However, it was significantly more difficult to express in the HOL proof system, requiring well over 100 proof tactics – no small task for a beginner to HOL. The overall proof script for reachability in sptrees contains well over 500 proof tactics and 50 definitions and theorems (including many lemmas about sptrees in general).

Specialisation for `closure_spt`

$$\frac{1.\ \texttt{WF}\ \mathit{start} \qquad 2.\ \texttt{WF_SET_TREE}\ \mathit{tree} \qquad 3.\ \texttt{domain}\ \mathit{start} \subseteq \texttt{domain}\ \mathit{tree}}{\texttt{domain}\ (\texttt{closure_spt}\ \mathit{start}\ \mathit{tree}) = \{a \mid \exists\, n.\ \texttt{isReachable}\ \mathit{tree}\ n\ a \wedge n \in \texttt{domain}\ \mathit{start}\}}$$

Figure 3.8: *The final result for the correctness proof of the abstract reachability functions, obtained by specialising the lemma for `close_spt` (*Figure 3.7*).*

The `closure_spt` function is a specialisation of the `close_spt` function, with arguments as follows: (*reachable* = `LN`), (*seen* = *start*). The lemma for `close_spt` (*Figure 3.7*) is therefore specialised by instantiating *reachable* and *seen* with these values, and setting *tree* = *fullTree* (in the first step of the recursion, the initial next-step function is equivalent to the recursive argument). Discharging tautologies in the assumptions gives the overall theorem for `closure_spt`, and the desired result for use in this project (*Figure 3.8*).

Chapter 4

Evaluation

This chapter summarises the main methodology adopted for evaluation (§ *4.1*), details of the core proof results obtained as part of primary evaluation (§ *4.2*), and finally a brief overview of secondary evaluation for FlatLang (§ *4.3*).

4.1 Evaluation methods

The primary evaluation method for this project is formal proof of correctness: if the optimisations are proved to always terminate and never affect observable behaviour, then they are unconditionally safe, and will fit into the CakeML compiler without adverse effects. This is therefore a success criterion for WordLang, and an extension criterion for FlatLang (this was achieved).

However, an "optimisation" which does not transform code is trivially safe, but is useless. The optimisation should therefore be shown to remove a significant amount of dead code. This is not feasible for WordLang: compiling to its stage in the pipeline and demonstrating code reduction is difficult due to the many stages prior to WordLang. The code reduction is also expected to be small due to the number of prior optimisation passes. The WordLang evaluation therefore falls primarily to the proof of correctness. However for FlatLang, demonstrating the efficacy of the optimisation using some example programs is a core aim of this project. This was only possible from May 7, 2018 due to delays in the CakeML development timeline (§ *2.2.1*). Note that this was not possible for arbitrary user-written code, but only the examples included in the CakeML codebase.

Further secondary evaluation metrics (such as the compilation speedup due to the optimisations) would then be considered, measured using the benchmarking suites in the CakeML codebase. However, this was not possible at the time of writing, as discussed in § *2.2.1*. The FlatLang branch of the repository (`type+module-update`) did not build, preventing compilation of user-written code and benchmarking.

4.2 Proof of correctness

This section describes the main evaluation of this project: the proofs of termination and semantic preservation for the optimisations in both WordLang and FlatLang.

The general approach to proving the termination of optimisation functions is in § *4.2.1*. The proofs of semantic preservation in WordLang (§ *4.2.2*) and FlatLang (§ *4.2.3*) follow, using the semantic definitions described in *Appendix A*.

Note that both proofs of correctness are contained within their respective languages. Unlike many proofs in verified compilers, they do not prove semantic preservation between intermediate languages, but that transformations within a language preserve semantics.

4.2.1 Proofs of termination

The optimisation must terminate on all input, to prevent changes to the behaviour of the overall compiler.

For all user-defined functions, HOL automatically attempts to prove their termination. If it fails, it will ask the user for a proof – this is more common for recursive functions with unclear termination cases. The function cannot be used until the termination proof is provided, and packaged up with the function definition.

Termination of recursive functions was proved by providing a measure function, which takes in the arguments of the function and returns a natural number. If this number strictly decreases on every recursive call, then clearly the function terminates. The core abstract reachability function (`close_spt` – see *Figure 3.6*) required such a proof, so it was modified to delete a node from its sptree next-step function supplied as its argument on each recursion. The measure for this is therefore the size of this next-step tree.

There is a subtlety with mutually recursive functions, required in the code analysis for FlatLang: as FlatLang expressions can contain single expressions and lists of expressions (*Figure A.3*), mutually recursive analysis functions are needed for the element case and the list case. These are declared in a single definition in HOL, and treated as a pair of functions – to prove termination, a measure for each function in the pair must be specified.

4.2.2 Proofs of semantic preservation in WordLang

The proof of correctness for WordLang is a core aim of this project. Due to the unforeseen complexity of working with such a low-level language, this was the longest part of the project and not the straightforward starter project initially planned.

As discussed above, testing the compiler through code inspection is not feasible, and benchmarking is unlikely to give any insight as the performance gains for WordLang are expected to be minimal. This proof is therefore the main evaluation method for WordLang, and so an important result in this project.

The rest of this subsection references material in § *A.1.2*, concerning WordLang semantics.

Lemma for WordLang semantic preservation

A more general lemma had to be proved before the desired theorem, as with the abstract reachability proofs (§ *3.3.3*). This lemma considers an arbitrary point in the evaluation of some WordLang program, consisting of a single current function, *function*, and its

accompanying state, *state*. This is compared to a point in the evaluation of a program with the same current function, but a modified state, *removedState*. These states are related by `word_state_rel`, and the evaluation cannot give an error (the WordLang semantics define this as failure of the program, and this optimisation is not concerned with failing programs – see § *A.1.2*). The proof shows that the two evaluations return the same result, and the resulting states are related in the same way as the original ones.

1. `evaluate` (*function*, *state*) = (*result*, *newState*)
2. *result* $\neq$ `SOME Error` 3. `word_state_rel` *reachable state removedState*
4. `noInstall` *program* 5. `noInstall_code` *state.code*
6. `noInstall_code` *removedState.code* 7. `codeClosed` *reachable state.code*
8. `domain` (`findWordRef` *function*) $\subseteq$ `domain` *reachable*
9. `gcNoNewLocs` *state.gc_fun*

$\exists$ *someState* . `evaluate` (*function*, *removedState*) = (*result*, *someState*) $\wedge$
`word_state_rel` *reachable newState someState* $\wedge$
`destResultLoc` *result* $\subseteq$ `domain` *reachable*

Figure 4.1: *The desired lemma for the proof of semantic preservation for* WordLang. *As before, variables are italicised and HOL functions are monospaced.*

The set of *reachable* WordLang locations is not explicitly generated by the optimisation functions as described in § *3.2.1*. This set instead fulfils all the necessary properties of the set that will be generated by the optimisation, and so is a more generalised version to simplify this proof. In particular, *reachable* is constrained to be at least as large as the set that will be generated by the optimisation (see below). The state *removedState* also emulates the state that will result from applying the optimisation.

The lemma is examined in more detail below.

Assumptions. Each assumption must be discharged when this lemma is specialised into the final result of semantic preservation for WordLang.

- **1.** This gives a handle on *result* and *newState*, the outputs of evaluating the initial WordLang program.
- **2.** The result of evaluation cannot be an error – in this case, the observable behaviour of the program would be failure, which should not occur for any well-formed WordLang program.
- **3.** The relation `word_state_rel`, parametrised by the sptree set *reachable*, relates two states that are equal in all but the *code* record field.

  For the *code* field, any function numbers in the *reachable* set must be common to both *state* and *removedState* (no reachable functions should have been removed).

  Finally, `word_state_rel` states that any WordLang locations in the *store*, *memory*, *stack*, or *locals* fields of *state* must be in the *reachable* set. To become

part of these fields, a location must have been encountered during execution and so is reachable. This assumption is necessary, as values can be retrieved from any of these fields during evaluation.

- **4, 5, and 6.** The update to the WordLang language on Apr 14, 2018[1], gave functionality to load and execute code at run time, which would break this optimisation. These assumptions ensure no `Install` functions are in the program to load external code.

- **7.** The *reachable* set must be at least as large as the one that will be generated by the optimisation. This assumption explicitly ties in with the optimisation implementation, and can be expanded:

$$\begin{aligned} &\texttt{codeClosed}\ \mathit{reachable}\ \mathit{state.code} \overset{\text{def}}{\Leftrightarrow} \\ &\qquad (\exists\ \mathit{codeList}\ .\ \mathit{state.code} = \texttt{fromAList}\ \mathit{codeList}) \wedge \\ &\qquad \big(\forall\ n\ m\ .\ n \in \texttt{domain}\ \mathit{reachable} \wedge \\ &\qquad \texttt{isReachable}\ (\texttt{mk_wf_set_tree}\ (\texttt{analyseWordCode}\ \mathit{code})\ n\ m) \\ &\quad \Rightarrow m \in \texttt{domain}\ \mathit{reachable}\big)\ . \end{aligned} \tag{4.1}$$

  This states that there is some associative list representing the the *code* field of the state – this is necessary as the optimisation acts on a list of functions, which is stored efficiently in an sptree in the state. The function `fromAList` trivially maps the list to the sptree, and is provided by HOL. The *reachable* set should then be closed under the `isReachable` relation (*Equation 3.1*) applied to the next-step function generated by code analysis of the program code. This gives the required constraint on the size of the *reachable* set.

- **8.** Any referenced locations in the current *function* must also be in the *reachable* set. When this lemma is specialised, the current function will be the standard starting function defined by the semantics, and this assumption will trivially hold.

- **9.** The garbage collector posed an unforeseen complication to this proof. It must not be able to introduce any new WordLang locations when it runs, or it would break this optimisation. This is clearly a reasonable assumption.

Conclusion. Evaluating the current *function* with *removedState* gives the same result as evaluation with *state*, and returns a resultant state *someState*. This resultant state must have *newState* related to it by `word_state_rel`, and any WordLang locations referenced in the result must be in the *reachable* set.

Only the first conjunct is important for semantic preservation. However, the other two are necessary in proving this lemma: the evaluation may call itself on another function number (such as in a `Call` to another function), and use the resulting intermediate value

[1]Commit URL: `https://github.com/CakeML/cakeml/commit/ffce1e9`.

and state to compute its final return value. These last two conjuncts therefore provide necessary results about these intermediate values and states in the proof by induction.

The proof strategy for this lemma is straightforward to understand, but difficult to execute, and was the single longest task of this project. There is an induction theorem associated with the evaluation function, which can be used to induct on the execution of the function. This splits it into twenty sub-cases, one for each type of WordLang `prog` (*Figure A.1*) – each sub-case must be proved separately, instantiating the inductive hypotheses as necessary. This is complicated by the various interactions with the state that occur, including machine-word operations and more. Proving that `word_state_rel` still holds for the resultant states after these operations was intricate, requiring many other significant lemmas.

This lemma was therefore difficult and time-consuming to tackle, requiring the creation of over 100 definitions and theorems, and using over 1500 proof tactics overall. For comparison, a variable liveness proof for DataLang (the language directly preceding WordLang) exists in the CakeML codebase, and it consists of a little over 50 definitions and theorems, and 500 proof tactics.

Specialisation into the final proof of WordLang semantic preservation

The lemma for WordLang correctness (*Figure 4.1*) is specialised into the final proof result by marrying it with the abstract reachability result (*Figure 3.7*), and instantiating terms to match the WordLang semantics (§ *A.1.2*).

$$\frac{\begin{array}{c} 1.\ \mathit{startingFunction} = \texttt{Call NONE (SOME } \mathit{start}\texttt{) [0] NONE} \\ 2.\ \texttt{evaluate}\ (\mathit{startingFunction},\ \mathit{state}) = (\mathit{result},\ \mathit{newState}) \\ 3.\ \mathit{result} \neq \texttt{SOME Error} \qquad 4.\ \mathit{state.code} = \texttt{fromAList}\ \mathit{originalCode} \\ 5.\ \mathit{reachable} = \texttt{closure_spt (insert } \mathit{start}\ \texttt{LN)} \\ \hookrightarrow \texttt{mk_wf_set_tree (analyseWordCode } \mathit{originalCode}\texttt{)} \\ 6.\ \mathit{newCode} = \texttt{removeWordCode}\ \mathit{reachable}\ \mathit{originalCode} \\ 7.\ \texttt{ALL_DISTINCT (MAP FST } \mathit{originalCode}\texttt{)} \\ 8.\ \texttt{domain (findLocState } \mathit{state}\texttt{)} \subseteq \texttt{domain}\ \mathit{reachable} \\ 9.\ \texttt{gcNoNewLocs}\ \mathit{state.gc_fun} \qquad 10.\ \texttt{noInstall_code}\ \mathit{state.code} \end{array}}{\begin{array}{c} \exists\ \mathit{someState}\ .\ \texttt{evaluate}\ (\mathit{startingFunction}, \\ \hookrightarrow \mathit{state}\ \texttt{with}\ \mathit{code} := \texttt{fromAList}\ \mathit{newCode}) = (\mathit{result},\ \mathit{someState}) \end{array}}$$

Figure 4.2: *The final result for the proof of* WordLang *semantic preservation, tying in the abstract reachability functions with the code analysis and the lemma shown in* Figure 4.1.

The *reachable* set from the lemma is instantiated to a set explicitly generated by the optimisation, and *removedState* is defined as the state resulting from code removal on the original state. The lemma is specialised to consider the start of the evaluation of

a WordLang program as defined by the semantics, rather than an arbitrary point in execution. The current function becomes a specific `Call` function to a supplied program entry point, which is also used by the abstract reachability functions as an argument (§ *3.2.1*).

The desired result is shown in *Figure 4.2*. Though it seems more complicated than the original lemma, it is in fact more specialised for application to this optimisation. In particular, the `word_state_rel` assumption has been almost discharged as a tautology: its only remnant is in the eighth assumption (see below). An overview of this final theorem follows.

Assumptions. These are simply specialisations of the assumptions in the lemma (*Figure 4.1*), with some discharged as tautologies.

- **1 and 2.** The evaluation in the lemma is specialised to match the WordLang semantics, with a specific starting function (*startingFunction*) consisting of a `Call` to a specific *start* function.
- **3.** As before, the evaluation should not give an error.
- **4.** As before, the program code can be transformed from an sptree into a list.
- **5 and 6.** The optimisation implementation is explicitly used here. The *reachable* state is defined as being generated by the abstract reachability function `closure_spt` (§ *3.3*), with the next-step function generated by code analysis through `analyseWordCode` (§ *3.2.1*). The function `mk_wf_set_tree` is as described in § *3.3.3*, *Equation 3.2*. The resulting code (*newCode*) in the conclusion is now defined as the result of the optimisation.
- **7, 8, and 9.** These cannot be discharged here, as they will only hold true in the context of the compilation pipeline – this is a proof about general WordLang programs, rather than compiler-generated WordLang programs.

  The seventh assumption states that all function numbers in the code are distinct. This will hold for compiled code, as the compiler will not duplicate any function numbers.

  The eighth is the last remnant of `word_state_rel`: all WordLang locations in the *store*, *memory*, *stack*, and *locals* fields of the *state* must be reachable. An unforeseen problem in this proof is that the initial state of a WordLang program is derived from DataLang (the preceding language in the compilation pipeline) – proving this holds true is therefore out of the scope of this project, as it involves a different intermediate language.

  The last assumption is as in the lemma: the garbage collector cannot create new WordLang locations. Once again, this is not provable without results from DataLang and the CakeML garbage collection process, and is out of the scope of this project.

Both of these unproved assumptions are expected to hold true, but their proof is intricate and not a task for this project. They will be proved in future CakeML work.

- **10.** As in the lemma (*Figure 4.1*), no `Install` functions are permitted – these would load and execute external code, breaking this optimisation. Each program must therefore be tested for this type of function before the optimisation is executed.

Conclusion. Evaluating the same starting function with the optimised code should give the same result as evaluation with the non-optimised code. As before, the `fromAList` function trivially maps an associative list to an sptree. This is clearly the final, powerful result of semantic preservation desired.

The proof of this theorem required results from all parts of the project: the code analysis functions, the abstract reachability analysis, and the WordLang lemma above. Each stage must output terms that satisfy the assumptions of the next – for example, the code analysis functions must be shown to output sptrees with the `WF_SET_TREE` property (this is easily achieved through use of `mk_wf_set_tree`).

4.2.3 Proofs of semantic preservation in FlatLang

The proof of semantic preservation in FlatLang was an extension aim of this project, and was completed. Due to the unforeseen difficulty of the WordLang proofs, limited time was available to execute this extension.

The rest of this subsection references material in § *A.2.2*, concerning FlatLang semantics.

Lemma for FlatLang semantic preservation

As in the WordLang case, a more general lemma must first be proved, and this follows a similar pattern to the WordLang lemma. It considers an arbitrary point in the evaluation of a FlatLang program, consisting of a list of declarations (*decs*, of type `(dec list)` – see § *A.2.1*), an environment (*env*), and state (*state*) – see § *A.2.2*. This is compared to the evaluation of a modified code list, with the same environment and a modified state. The relation `flat_state_rel` relates the original and modified states, and the evaluation cannot return a type error. Both evaluations should return the same result (*result*) and set of new constructors (*newCons*), and the two returned states should be related by `flat_state_rel` in the same way.

Note the primary difference to the WordLang lemma: in WordLang, program code is stored as part of the program state, whereas FlatLang has separated state and code. A relation between the original and modified code must therefore also be specified (they are related by `removeUnreachable`). The overall lemma is shown in *Figure 4.3*.

$$\frac{\begin{array}{c}1.\ \texttt{evaluate_decs}\ env\ state\ decs = (newState,\ newCons,\ result)\\ 2.\ result \neq \texttt{Rabort Rtype_error}\\ 3.\ env.exh_pat \quad 4.\ \texttt{flat_state_rel}\ reachable\ state\ removedState\\ 5.\ \texttt{removeUnreachable}\ reachable\ decs = removedDecs\\ 6.\ \texttt{decsClosed}\ reachable\ decs\\ 7.\ \texttt{domain}\ (\texttt{findEnvGlobals}\ env) \subseteq \texttt{domain}\ reachable\end{array}}{\begin{array}{c}\exists\ someState\ .\\ \texttt{evaluate_decs}\ env\ removedState\ removedDecs = (someState,\ newCons,\ result) \wedge\\ \texttt{flat_state_rel}\ reachable\ newState\ someState \wedge\\ \texttt{domain}\ (\texttt{findResultGlobals}\ result) \subseteq \texttt{domain}\ reachable\end{array}}$$

Figure 4.3: *The desired lemma for the proof of semantic preservation for* FlatLang*. As before, variables are italicised and HOL functions are monospaced.*

Assumptions. These largely follow a similar pattern to the WordLang case.

- **1 and 2.** Handles on *result* and *newState* are provided, and the evaluation cannot produce a type error.
- **3.** All pattern matches in the input FlatLang program (such as in `Mat` pattern-match expressions or `Handle` exception-handler expressions) must be exhaustive.

  This was an unforeseen necessity: without this constraint, a pattern-match can run out of patterns and give an error, causing problems in the proof. As a result, the optimisation must be executed marginally later in the compilation pipeline than desired, after some other FlatLang optimisations which ensure exhaustive pattern-matches. This is a small drawback, as those prior passes will execute on code that will then be removed by this optimisation (this is wasted work). This is another trade-off between implementation efficacy and simplicity of proof for verified compilers, and can be seen as a phase order problem affecting the proof of correctness.
- **4.** The relation `flat_state_rel` mirrors the WordLang relation, `word_state_rel`. All fields of the *state* and *removedState* records are equal, except the list of *globals* (the optimisation acts over global variables, so naturally the two states will store different globals).

  A separate relation is defined for the globals lists: for any global variable in the *reachable* set, the two lists must have the same values (reachable global variables should not be changed). Furthermore, any variables present in *removedState* must also be in *state* (no new global variables can be introduced by the removal). Note this does not preclude *state* storing a variable where *removedState* does not (i.e. because it has been removed), as desired.

  Lastly, any global variables referenced in the *refs* field of the *state* must also be in the reachable set – no variables will become part of this field during execution unless they are reachable.

- **5 and 6.** The *reachable* set must be related to the code lists, *decs* and *removedDecs*.

  The fifth assumption states that *removedDecs* is a version of *decs* filtered to remove all code without observable behaviour. This means that all impure declarations are kept (§ *3.2.2*), and any declaration which initialises a reachable global variable is kept.

  The sixth assumption constrains the *reachable* set to be at least as large as the set that will be generated by the optimisation. The definition of `decsClosed` can be expanded:

$$
\begin{aligned}
&\texttt{decsClosed}\ \mathit{reachable}\ \mathit{decs} \overset{\text{def}}{\Leftrightarrow} \texttt{analyseCode}\ \mathit{decs} = (\mathit{startingSet},\ \mathit{nextStep}) \\
&\qquad \Rightarrow \texttt{domain}\ \mathit{startingSet} \subseteq \texttt{domain}\ \mathit{reachable} \wedge \\
&\qquad\ \big(\forall\ n\ m\ .\ n \in \texttt{domain}\ \mathit{reachable} \wedge \\
&\qquad\ \texttt{isReachable}\ (\texttt{mk_wf_set_tree}\ (\texttt{analyseCode}\ \mathit{decs})\ n\ m) \\
&\qquad\ \Rightarrow m \in \texttt{domain}\ \mathit{reachable}\big)\ .
\end{aligned}
\tag{4.2}
$$

  This is very similar to the WordLang case; the sole difference is that the FlatLang code analysis determines the starting set of immediately reachable nodes (*startingSet*), rather than having it supplied as an argument. The starting set must therefore be a subset of the set of reachable nodes too.

- **7.** The environment may contain references to global variables during program execution, which must all be in the set of reachable nodes.

Conclusion. Evaluating the optimised code (*removedDecs*) with *removedState* gives the same result and the same new constructors as evaluation of the original code (*decs*) with *state*. The resultant state generated (*someState*) must have *newState* related to it by `flat_state_rel`, and any global variables referenced in the result must be in the *reachable* set. As in the WordLang lemma, only the first conjunct is of interest for semantic preservation, but the others allow for stronger inductive hypotheses in the proof.

The proof strategy for this lemma is more intricate than in the WordLang case, though the proofs themselves are simpler. This is due to the definition of the `evaluate_decs` function: this first calls the function `evaluate_dec` (NB: singular `dec`) on the first element of the list of declarations, and then recursively calls itself on the remainder of the list (if no error is returned). The `evaluate_dec` function simply adds new constructors to the output for exception or type declarations, but for value declarations will call a recursive `evaluate` function on the expression contained within the declaration.

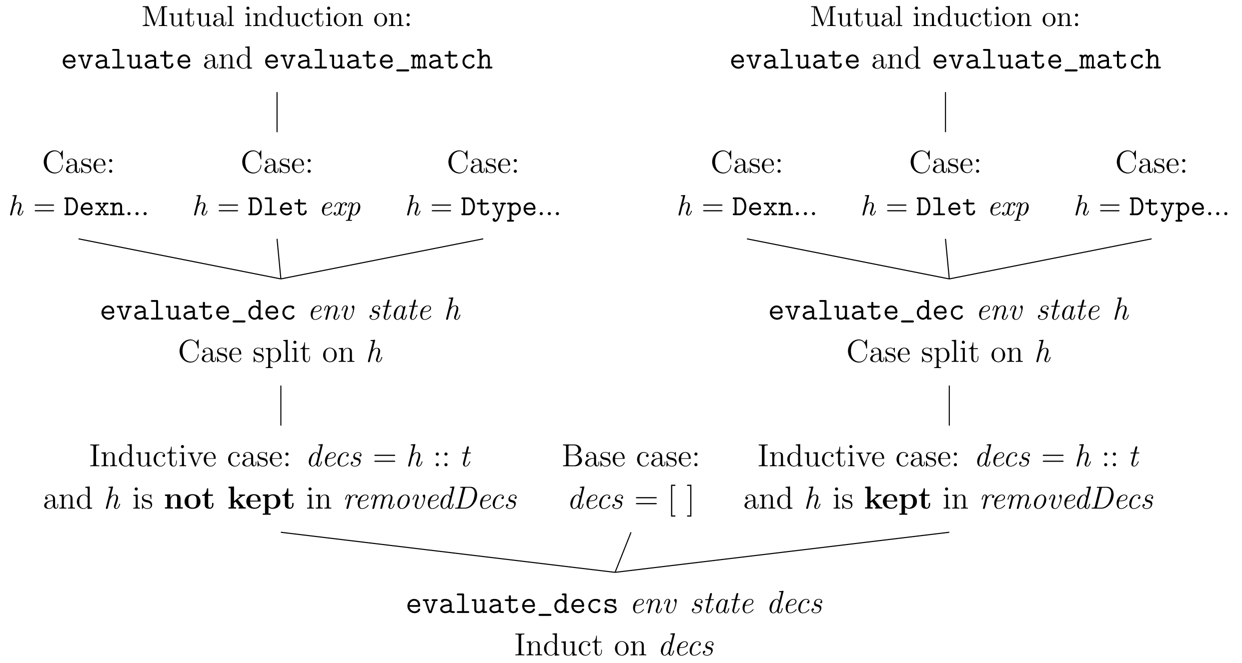


Figure 4.4: *The proof strategy for the* FlatLang *semantic preservation lemma, shown as a tree. The root (lowermost) node represents the overall goal, and the children represent the various case splits. The collected proofs of all leaves together enable the proof of the root.*

This tiered evaluation function necessitates a tiered proof structure too, shown in *Figure 4.4*.

At the first level (`evaluate_decs`), induction on the list of declarations splits the proof into the empty list case ([ ]) which can be discharged here, and the constructor case ($h :: t$). This case then further splits based on whether the first declaration (h) is kept by the optimisation or removed.

Each of these two cases then examines `evaluate_dec` at the second level of proof, applied to the first declaration, h. Any exception or type declarations can be discharged, leaving only value declarations containing an expression (of the form `Dlet` *exp*).

The third level of proof then considers the `evaluate` function applied to this expression – the function is mutually recursive with the `evaluate_match` function, concerning pattern matching. The proof therefore follows a mutual induction using the induction theorem generated by the definition of these two functions: this splits the proof into a case for each FlatLang expression or pattern-match expression.

The proof of this lemma required over 150 definitions and theorems, using over 1500 tactics altogether.

Specialisation into final proof of FlatLang semantic preservation

The lemma must be specialised to explicitly generate the *reachable* set using the optimisation. The state and environment must be specialised to the initial values as specified by the semantics of FlatLang too. This result is shown in *Figure 4.5*.

$$\frac{\begin{array}{c}1.\ \texttt{evaluate_decs}\ \mathit{initialEnv}\ \mathit{initialState}\ \mathit{decs} = (\mathit{newState},\ \mathit{newCons},\ \mathit{result}) \\ 2.\ \mathit{result} \neq \texttt{Rabort Rtype_error} \\ 3.\ \mathit{initialEnv.exh_pat} \qquad 4.\ \texttt{analyseCode}\ \mathit{decs} = (\mathit{startingSet},\ \mathit{nextStep}) \\ 5.\ \texttt{closure_spt}\ \mathit{startingSet}\ (\texttt{mk_wf_set_tree}\ \mathit{nextStep}) \\ 6.\ \texttt{removeUnreachable}\ \mathit{reachable}\ \mathit{decs} = \mathit{removedDecs}\end{array}}{\begin{array}{c}\exists\ \mathit{someState}\ . \\ \texttt{evaluate_decs}\ \mathit{env}\ \mathit{state}\ \mathit{removedDecs} = (\mathit{someState},\ \mathit{newCons},\ \mathit{result})\end{array}}$$

Figure 4.5: *The final result of semantic preservation for the optimisation in* FLATLANG.

This is similar to the FLATLANG lemma above, however the *reachable* set is explicitly derived, and the assumption about `flat_state_rel` is discharged as a tautology. A brief overview of this theorem follows:

Assumptions. These are very similar to those in the FLATLANG lemma.

- **1, 2, and 3.** These are unchanged from the lemma, except the *state* and *env* have become the initial state (*initialState*) and environment (*initialEnv*) specified by the FLATLANG semantics. The initial state contains no global variables and its *refs* field is empty, and the initial environment has an empty *v* field of values. This trivially discharges many assumptions about the state or the environment from the lemma.
- **4 and 5.** The generation of the *reachable* set is made explicit. The `analyseCode` function generates a starting set of immediately reachable nodes and a next-step function, and the `closure_spt` function determines an overall set of reachable nodes from these.
- **6.** This is unchanged from the lemma.

Conclusion. The conclusion keeps only the first conjunct of the lemma, as it is the only one necessary to show semantic preservation. It states that evaluating the optimised code (*removedDecs*) gives the same result as evaluating the original code (*decs*), with all else held constant.

This was proved using results from the code analysis functions, the abstract reachability analysis, and the FLATLANG lemma above: the results were stitched together in sequence, with each stage satisfying the assumptions of the next (in a similar way to the WORDLANG specialisation).

4.3 Secondary evaluation

Example programs provided in the CakeML codebase were used to demonstrate the resulting code reduction of the optimisation in FLATLANG. This analysis was not possible

until May 7, 2018[2] due to delays in the CakeML development timeline (§ *2.2.1*), as the example programs used did not build.

The results are summarised in *Figure 4.6* (raw data can be found in *Figure C.1*). The evaluation considered eight example programs (including the UNIX tools `cat`, `diff`, `echo`, `grep`, `patch`, and `sort`), and evaluated five measures of code size on each, both before and after optimisation, giving a measure of how much code was removed.

% of code removed for various measures of code size	No. of top-level declarations	Expression size	No. of global variable initialisations	No. of global variable lookups	Term size
Mean	81%	77%	83%	78%	83%
Std. deviation	9%	18%	9%	15%	14%

Figure 4.6: *Summarised results of code reduction evaluation of the optimisation in* FlatLang. *Five measures of code size were evaluated on eight example programs. In all measures, the optimisation pass gives significant code reduction, demonstrating its efficacy and usefulness.*

The measures considered are as follows:

- **Number of top-level declarations.** Each declaration in FlatLang is of type `dec`, and FlatLang code is a `(dec list)` (§ *A.2.1*). This measure is effectively the size of that list.
- **Expression size.** Provided in the CakeML codebase are measures of expression size for FlatLang expressions – this is the number of constructors required to define the expression. Expressions can be contained within the value declarations over which the optimisation acts (§ *A.2.1*).
- **Number of global variable initialisations.** The number of `(GlobalVarInit : Op)` operations in the code, each of which initialises a global variable. This is therefore the number of global variables that the optimisation determined to be reachable.
- **Number of global variable lookups.** The number of `(GlobalVarLookup : Op)` operations in the code, each looking up the value of a global variable.
- **Term size.** This is an SML measure rather than a HOL-implemented one. It represents the size of a HOL term in terms of the number of SML constructors required to express it.

4.4 Summary of results

The optimisations in both WordLang and FlatLang have been proved to be safe using HOL: they will never affect the observable behaviour of any program. The FlatLang

[2] Commit URL: `https://github.com/CakeML/cakeml/commit/52ebb2`.

optimisation has also been shown to remove a significant proportion of code from some example programs.

The powerful proof results ensure that both optimisations can be integrated into the CakeML codebase without adversely affecting the rest of the compiler. The demonstrations in FlatLang show that the optimisation successfully ameliorates the problem it was designed to by removing much of the basis library dead code from test programs – it is therefore an effective addition to the CakeML compilation pipeline.

Chapter 5

Conclusion

This project has contributed to an open-source, optimising compiler for a high-level language, but importantly one that is also verified to be correct. Over the course of the project, the trade-off between efficacy of optimisation and ease of proof was encountered several times: when a proof was found to be too difficult, the optimisation had to be simplified to allow the proof to continue.

This chapter includes a full summary of the work completed (§ *5.1*), possible improvements that could be made on the project in future (§ *5.2*), and additional related work that could be attempted (§ *5.3*).

5.1 Summary of the work completed

I have successfully implemented two dead code elimination passes and verified them to preserve semantics: in WordLang (the antepenultimate intermediate language, and relatively low-level), and in FlatLang (the first intermediate language, and relatively high-level).

After a period of familiarisation with the HOL4 interactive theorem-prover, abstract reachability functions were implemented and proved correct: operating on a next-step function, these determine an overall reachable set of nodes. Code analysis functions were implemented for WordLang and FlatLang, to generate a next-step function over function numbers (for WordLang) or global variables (for FlatLang). Combining reachability functions and code analysis gave the overall optimisations, and these were evaluated by proving properties of semantic preservation for both. The FlatLang optimisation was further evaluated by measures of code reduction.

These optimisations were then integrated into mainstream CakeML development on the `type+module-update` branch. This contains the most up-to-date version of FlatLang, which does not exist on the `master` branch at the time of writing. Further evaluation measures such as benchmarking and regression testing were not possible due to the `type+module-update` branch not building at the time of writing.

Three out of the four core success criteria were achieved outright: the optimisation was implemented for FlatLang, both implemented and verified for WordLang, and demonstrated to remove unused FlatLang code. The final criterion was achieved as far

as possible: the FlatLang optimisation is not part of the latest version of CakeML, but is a part of the latest version that contains FlatLang. It will be a part of the main compiler as soon as FlatLang itself is integrated into the `master` branch.

An extension aim of the project was achieved: the FlatLang optimisation pass was verified to preserve semantics. Another extension aim was not fully achievable: empirical tests of the compilation speedup due to the FlatLang optimisation were not possible as the compiler did not build at the time of writing, an issue out of the control of this project. However, some empirical measures of code size reduction were used to test the optimisation.

5.2 Improvements on the work completed

There are several potential areas in which the optimisations could be improved to give performance or efficacy benefits. Some of these areas are highlighted below.

- **Granularity of the FlatLang optimisation.** The high-level nature of FlatLang makes it difficult to determine if a piece of code can have an observable effect. For this project, many approximations had to be made, categorising code coarsely on purity and whether it is "hidden" (§ *3.2.2*). Improving the granularity of this categorisation could give a finer optimisation, allowing more dead code to be eliminated. However, this will come at the expense of a significantly more difficult proof of correctness.

  The optimisation also over-approximates by assuming that in any expression, any initialised global variables can look up all global variables referenced. Some form of flow analysis could improve the efficacy of the optimisation.

- **Handling of `Install` functions in the WordLang optimisation.** Late on in the project, the WordLang language was updated with the functionality to load and run new code at run time. This can cause problems when used with the optimisation in this project: code that was previously considered dead (and has been removed) may be referenced by the code newly installed at run time. This was handled by modifying the optimisation not to run on any WordLang program containing an function to install code at run time – this could be improved by running the analysis pass, and then only executing the code removal if no such `Install` functions are determined to be reachable.

- **Necessity of the `mk_wf_set_tree` function.** The `mk_wf_set_tree` function (§ *3.3.3*) is a costly part of the optimisation execution. It may not be necessary, as compiled WordLang or FlatLang programs may result in next-step functions satisfying the `WF_SET_TREE` predicate. The predicate encapsulates the concept that WordLang or FlatLang code should not reference any function numbers or global variables (respectively) that have not yet been defined. Proving that this is the case would eliminate the need for this function, and make the optimisation more efficient.

5.3 Further work

Some areas for future work have been highlighted over the course of the project, where the project could be built on or applied. Some of these are summarised below.

- **Proof of properties of the derived WordLang state.** The correctness proof for the WordLang optimisation relies on results about the initial state of a WordLang program, in particular the garbage collector and stored WordLang locations. These results are expected to hold, but as the state is derived from DataLang (the previous intermediate language in the compilation pipeline), the proof was out of the scope of this project. Proving these results is therefore a possible extension task – however, the complexity of garbage collection makes this a significant undertaking.

- **Further evaluation of the FlatLang optimisation.** As the `type+module-update` branch of the GitHub repository does not build, no empirical evaluation of the speedup due to the FlatLang optimisation was possible. Quantitative analysis using the benchmarking suites on the CakeML codebase would provide insights into the benefits of the optimisation, and highlight any areas for improvement.

- **Further optimisations for WordLang or FlatLang.** The knowledge of WordLang and FlatLang gained through this project could be transferred into writing and verifying another optimisation for either of these languages. For example, in FlatLang a possible optimisation is a global purity analysis pass. This would move pure constant computations out of loops, turning them into new declarations with fresh names.

Bibliography

[1] *The HOL System Description*, volume Kananaskis-11. 2017.

[2] *The HOL System Logic*, volume Kananaskis-11. 2017.

[3] CakeML development team. *CakeML*. `https://cakeml.org/`, accessed April 2018.

[4] Michael J. C. Gordon. *From LCF to HOL: a short history*. 1996.

[5] Tyler Hanna, Chester Holtz, and Jonathan Liao. *Comparative Analysis of Classic Garbage-Collection Algorithms for a Lisp-like Language*. University of Rochester, 2015.

[6] John Harrison, Josef Urban, and Freek Wiedijk. *History of interactive theorem proving*.

[7] INRIA. *CompCert - The CompCert C Compiler*. `http://compcert.inria.fr/compcert-C.html`, accessed April 2018.

[8] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. *CakeML: A Verified Implementation of ML*. Symposium on Principles of Programming Languages, 2014.

[9] John McCarthy and James Painter. *Correctness of a Compiler for Arithmetic Expressions*, volume 19 of *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[10] Magnus O. Myreen and Scott Owens. *Proof-producing Synthesis of ML from Higher-Order Logic*. International Conference on Functional Programming, 2012.

[11] Thomak Tuerk and PROSPER group. *Interactive Theorem Proving (ITP) Course*. 2017. Adapted from a course given for advanced master students given at KTH, Stockholm. `https://hol-theorem-prover.org/hol-course.pdf`, accessed April 2018.

Appendix A

Intermediate languages

This appendix gives a basic overview of the intermediate languages WordLang and FlatLang, on which the optimisations written for this project must act. An understanding of the syntax, structure, and semantics of these intermediate languages is vital to the execution of the project: in order to ensure preservation of semantics, these semantics must be well understood. The code analysis driving the safety of the optimisation must operate on the language syntax, with a clear understanding of the structure of a program to prevent incorrect analysis too. An overview of WordLang can be found in § *A.1*, and an overview of FlatLang in § *A.2*.

A.1 WordLang

The intermediate language WordLang is the antepenultimate language in the CakeML compilation pipeline. It is an imperative language, with machine words, memory, a stack, and a garbage collection primitive. Its simple structure makes it a good candidate for a starter project tackling dead code elimination.

This section describes the structure and semantics of a WordLang program.

A.1.1 Structure of a WordLang program

A WordLang program consists of a list of tuples of type `(num * num * prog)` (a simplified `prog` data type is shown in *Figure A.1*). The first number of the tuple is the function number (the WordLang location storing that particular function), while each `prog` roughly corresponds to an imperative function, and can reference other functions by their numbers. The second number in the tuple is the number of arguments accepted by the function – it is not relevant to the optimisations implemented in this project.

```
datatype 'a prog =
    Skip                                                       (* skip *)
  | Move of ...                                (* move local variables *)
  | Inst of ...                               (* assign local variable *)
  | Assign of ...      (* assign local variable to value of another *)
  | Get of ...                                   (* get local variable *)
  | Set of ...                                   (* set local variable *)
  | Store of ...                                    (* store in memory *)
  | MustTerminate of prog                          (* must not time-out *)
  | Call of                                          (* call a function *)
        (return information) * target * arguments * handler
  | Seq of prog * prog                          (* sequential execution *)
  | If of ... * prog * prog                             (* if statement *)
  | Alloc of ...                                    (* allocate storage *)
  | Raise of ...                                   (* raise an exception *)
  | Return of ...                                             (* return *)
  | Tick                                                  (* clock tick *)
  | LocValue of v * l   (* for variable v, location l: assign v := l *)
  | Install of ...                           (* install code at run time *)
  | CodeBufferWrite of ...                      (* write to code buffer *)
  | DataBufferWrite of ...                      (* write to data buffer *)
  | FFI of ... ;                           (* foreign function interface *)
```

Figure A.1: *The data type for the* WordLang *function. For simplicity, most constructor arguments are omitted, save for recursive cases or for arguments of particular interest to the analysis. Within the* `(return information)` *and* `handler` *arguments to the* `Call` *constructor, there are also recursive instances of the* `prog` *data type, to allow for return handler and exception handler code respectively.*

Note that the `Install`, `CodeBufferWrite`, and `DataBufferWrite` expressions were added by the CakeML team on Apr 14, 2018[1] as part of an update to the WordLang language, giving functionality to load and run code at run time. This required some modifications to the proof of correctness for the WordLang optimisation implemented in this project (§ *4.2.2*).

A.1.2 Semantics of WordLang

The semantics of WordLang specify an evaluation function, which takes in a WordLang function of type `prog` and a WordLang *state*, and outputs a result as well as a resulting state. The result can be either a time-out, exception, value, or even no result at all.

The WordLang state is a record data type, shown in *Figure A.2*, and encapsulates the state of the whole program – this includes values stored in memory, the stack, the remaining code of the program (other than the single function that is the first argument

[1] Commit URL: `https://github.com/CakeML/cakeml/commit/ffce1e9`.

of the evaluation function), local variables, a clock, and more besides. The complexity of this state as well as the low-level nature of WordLang made the proof of semantic preservation for WordLang much more difficult than expected. For example, uses of machine words (including bit operations) and references to locations stored in the state further complicated the analysis of exactly what is reachable, and so made it difficult to prove that no reachable code had been removed by the optimisation.

```
datatype ('a, 'c, 'ffi) state =
 <| locals  : ('a word_loc) sptree                    (* local variables *)
  ; fp_regs : num |-> word64                (* floating-point registers *)
  ; store   : store_name |-> 'a word_loc
                     (* a store containing pointers to the heap and more *)
  ; stack   : ('a stack_frame) list                         (* the stack *)
  ; memory  : 'a word -> 'a word_loc
                          (* a function representing memory mappings *)
  ; mdomain : ('a word) set            (* domain of the memory function *)
  ; permute : num -> num -> num     (* sequence of bijective mappings *)
  ; compile : ...     (* compile function for code loaded at run time *)
  ; compile_oracle : ...
                        (* compile oracle for code loaded at run time *)
  ; code_buffer : ...      (* code buffer for code loaded at run time *)
  ; data_buffer : ...      (* data buffer for code loaded at run time *)
  ; gc_fun  : 'a gc_fun_type        (* the garbage collector function *)
  ; handler : num         (* position of current handle frame on stack *)
  ; clock   : num                                           (* the clock *)
  ; termdep : num
        (* number of MustTerminate functions that can still be entered *)
  ; code    : (num * ('a prog)) sptree              (* the program code *)
  ; be      : bool                   (* true if words are big-endian *)
  ; ffi     : 'ffi ffi_state            (* foreign function interface *)
  |> ;
```

Figure A.2: *The record data type for* WordLang *program state. Of particular interest to the analysis for this project are the* `locals`*,* `store`*,* `stack`*, and* `memory`*. These can contain items of type* `('a word_loc)`*, which are either machine words or* WordLang *locations – these are the very locations over which the analysis operates to determine code reachability.*

Note that the `compile`, `compile_oracle`, `code_buffer`, and `data_buffer` fields were added by the CakeML team on Apr 14, 2018[2] as part of an update to the WordLang language, giving functionality to load and run code at run time. This required some modifications to the proof of correctness for the WordLang optimisation implemented in this project (§ *4.2.2*).

[2] Commit URL: `https://github.com/CakeML/cakeml/commit/ffce1e9`.

The semantics of WordLang are then defined in terms of the observable behaviour of a program– possible semantics are failure, the return of a value, or the non-termination of the program. A semantics function encapsulates this: it takes as arguments a start location (the location of the first WordLang function to execute) and a WordLang state, and returns the observable behaviour of the program. This is computed by running the evaluation function described above, with a specific first function (a `Call` to the provided starting location) and the provided state as arguments. If the evaluation returns a time-out or runs out of space, the behaviour is the program is deemed to be failure. If not, the semantics function examines the resulting state's foreign function interface, and uses this to determine if the evaluation terminates or not – the program behaviour can then be classified as terminating or diverging.

A.2 FlatLang

The intermediate language FlatLang is the first language in the compilation pipeline – the abstract syntax tree is compiled directly to it. All global scoping is resolved, and all modules are removed. Each value definition is also given a slot in a global variable store – it is these global variables over which the dead code elimination analysis will be performed. However, the resolution of modules and global scoping results in the entire basis library being prepended onto the user-written source code – this is true even for the simplest "hello world" programs, and is obviously wasteful and unnecessary, greatly bloating code and creating further work for translators between intermediate languages. This therefore makes FlatLang a good candidate for dead code elimination. As FlatLang is so early in the compilation pipeline too, such an optimisation could significantly reduce the overhead of compilation further along the pipeline by preventing unnecessary code being compiled further.

This section describes the structure and semantics of a FlatLang program. Note that FlatLang exists only on the `type+module-update` branch of the CakeML GitHub repository (§ *2.2.1*).

A.2.1 Structure of a FlatLang program

A FlatLang program consists of a list of declarations of values, types, and exceptions – these declarations are of type `dec`. For dead code elimination, only the value declarations are of interest, and these contain FlatLang expressions. Expressions are of type `exp`, and have parallels to CakeML expressions (which in turn are similar to SML expressions). Each expression also contains a trace of the expression through the compiler, allowing for exploration of some of the transformations. Operations of type `Op` are also defined, but only two of these are of interest for this analysis – the initialisation of a global variable (`GlobalVarInit`), and lookup of a global variable (`GlobalVarLookup`). Simplified definitions of operations, expressions, and declarations are shown in *Figure A.3*.

```
1  (* Op = global variable init | global variable lookup | ... *)
2  datatype Op = GlobalVarInit num | GlobalVarLookup num | ...;
3
4  datatype exp =
5      Raise of exp                                   (* raise exception *)
6    | Handle of exp * ((... * exp) list)            (* handle exception *)
7    | Lit of ...                                           (* literal *)
8    | Con of ... * (exp list)                          (* constructor *)
9    | Var_local of ...                              (* local variable *)
10   | Fun of ... exp                           (* function definition *)
11   | App of Op * (exp list)                 (* operation application *)
12   | If of exp * exp * exp                           (* if statement *)
13   | Mat of exp * ((... * exp) list)                (* pattern match *)
14   | Let of ... * exp * exp                        (* let expression *)
15   | Letrec of ((... * exp) list) * exp;        (* letrec expression *)
16
17 (* dec = value declaration | type decl. | exception decl. *)
18 datatype dec = Dlet of exp | Dtype of ... | Dexn of ...;
```

Figure A.3: *The simplified data types for* FlatLang *operations (`Op`), expressions (`exp`), and declarations (`dec`). For the purposes of this project, only the `GlobalVarInit` and `GlobalVarLookup` operations are of interest, and only the `Dlet` value declaration. Expression traces in the expression data type are omitted, as are any non-recursive arguments other than those of type `Op` (as in the `App` case). Note that a* FlatLang *program is of type `dec list`.*

A.2.2 Semantics of FlatLang

Like the WordLang semantics, the FlatLang semantics define an evaluation function. In this case, the function takes in as arguments an environment, a FlatLang state, and FlatLang code (a list of top-level declarations, of type `(dec list)`). It returns a resulting state, a set of any new constructors the program has created, and an option type representing the outcome of evaluation (if any).

The state and environment arguments are record data types, and are shown in *Figure A.4*. They define the overall state of the program, including all local and global variables, as well as some whole-program parameters such as whether pattern matches are required to be exhaustive – this parameter in particular was an important constraint for the correctness proof for the FlatLang optimisation (§ *4.2.3*). The state is far simpler than the WordLang case, making it easier to handle in proofs. However the environment can store local values which may reference global variables, complicating the proof that no reachable global variables have been removed.

```
1  datatype 'ffi state =
2   <| clock    : num                                          (* the clock *)
3    ; refs     : value store                            (* a store of values *)
4    ; ffi      : 'ffi ffi_state                  (* foreign function interface *)
5    ; globals : (value option) list
6                       (* a list of global variables, indexed by number *)
7    |> ;
8
9  datatype environment =
10  <| v          : (variable_name, value) alist
11                            (* a list of local values, indexed by name *)
12   ; c          : ((id , type_id), arity) set
13                                  (* the set of existing constructors *)
14   ; exh_pat    : bool  (* true if pattern matches must be exhaustive *)
15   ; check_ctor : bool       (* true if constructors must be declared *)
16   |> ;
```

Figure A.4: *The record data types for* FlatLang *program state and* FlatLang *environment. The global variables considered in this optimisation are numbers, and the values they store are held in a list in the* `globals` *field of the state – the* nth *element of the list corresponds to the* nth *global variable, and is* `NONE` *if not yet initialised. References to global variables (through* `(GlobalVarInit : Op)` *and* `(GlobalVarLookup : Op)`*) can occur in any values – therefore the* `refs` *and* `globals` *fields of the state, as well as the* `v` *field of the environment can contain references.*

The FlatLang semantics are defined in terms of the observable behaviour of the program, just as in WordLang. The semantics function for FlatLang takes as arguments a foreign function interface, the `exh_pat` and `check_ctor` flags of the environment, and a program in the form of a list of declarations (of type `dec list`). It calls the evaluation function with an initial environment, an initial state, and the program, and if a type error ensues, the behaviour of the program is failure. Otherwise the resulting state (in particular, its foreign function interface) and returned outcome are used to determine if the observable behaviour is termination or divergence.

Appendix B

HOL4

This appendix details the core concepts of the HOL4 interactive theorem-prover necessary for this project [1] [2]. This includes a basic understanding of HOL higher-order logic constructs (§ *B.1*), descriptions of the main interfaces with HOL over the project (§ *B.2* and § *B.3*), and some of the proof tools used (§ *B.3.1* and § *B.3.2*).

B.1 Basic principles of HOL

The three basic HOL data types are types, terms, and theorems. These can be constructed and destructed by the SML functions provided by the HOL library. An understanding of these core data types was therefore necessary to understand HOL operation.

HOL types. Type variables or type operators – simple types such as `bool` are just type operators which take no arguments, and compound types (such as function types) can be constructed too. Simple type operators such as `bool` and $\rightarrow$ (the function type operator) are bound to type operators of the same name in SML to enable easy construction – note the difference between the SML handle and the underlying representation in HOL.

HOL terms. Constants, variables, λ-abstractions, and combinations (function applications). Variables and constants must have (possibly polymorphic) types specified. Constants must also be scoped into theories, and λ-abstractions are equivalent up to α-conversion. Each term has a type. Terms are only equivalent if both their type and the underlying term are equivalent. This can cause problems in proof when the same syntactic term has different polymorphic types.

HOL theorems (`thm`). Proven theorems – (*hypotheses* $\vdash$ *conclusion*) in standard logical notation, where both *hypotheses* and *conclusion* are of type `bool`. These can only be created by a trusted module (in the LCF tradition – see § *1.2.1*), by calling its functions. A proof therefore exists for any theorem created, consisting of a sequence of calls to those functions, which encapsulate primitive rules of inference: assumption introduction and discharge, reflexivity, abstraction and β-reduction, parallel substitution,

type instantiation, and *modus ponens*.

The quotation parser (again taking after LCF) allows construction of types and terms without using explicit (and tedious) SML functions. Anything enclosed in backticks (" ``...`` ") will be parsed to return a type or a term, enabling user-friendly construction of higher-order logic entities[1]. Type inference, name resolution, and overloading resolution are also executed on terms specified in this way. This is a layer that HOL provides above the underlying SML, rather than SML syntax – it is handled by the HOL parser.

HOL's key strength is that it abstracts away from low-level proof structures, enabling high-level reasoning about the overall proof strategy. This is achieved through automation of many proofs – for example, HOL will automatically try to prove that recursive functions terminate (raising an error if it cannot, and asking for a proof to be provided), and automatically generate induction theorems for recursively-defined data structures and functions. Understanding how to store and use these generated theorems was an important skill to learn.

HOL operates on a higher-order logic of expressions and types, known as simple type theory [2]. First-order logic constructs (such as Boolean connectives and quantifiers) are defined, but have no special significance (unlike in first-order logic). For example, Booleans are simply constants of type `bool` in HOL which must be defined, and all properties stated as axioms or derived rules. For example, `T` (*true*) and `F` (*false*) are specified as distinct values in `boolTheory`[2].

B.2 Scripts and theories

Managing a large project such as CakeML requires directories of HOL scripts, managed by a `Holmakefile` in each directory. HOL provides the `Holmake` tool, which "compiles" and "executes" a script – it compiles the SML, running the quotation parser on any HOL code, and running through all (valid) proofs to generate theorems. It returns a theory file, which can be reused in other projects. `Holmake` uses a `Holmakefile` to manage dependencies, (re-)compiling scripts as necessary when they are updated. Familiarity with this tool was therefore necessary for this project. This is analogous to the GNU `make` tool, which uses a `makefile` to control the generation of executables.

HOL scripts allow HOL definitions and theorems to be pre-written, and imported into the REPL later. Using only the read-evaluate-print-loop (REPL) to interact with HOL is unwieldy.

Theories contain sets of types, constants, definitions, and axioms. They also contain a set of all theorems proved so far from the axioms and definitions, and are the result of any HOL session (HOL always maintains a current theory of the work done so far). The `Holmake` tool exports theories to disk, allowing for persistent storage of definitions and theorems.

[1] For example, we can write ``λx.x`` to denote the identity function, rather than write a complicated mess of constructor functions.

[2] The `BOOL_EQ_DISTINCT` theorem: $\vdash\ \neg(\mathtt{T} \Leftrightarrow \mathtt{F})\ \wedge\ \neg(\mathtt{F} \Leftrightarrow \mathtt{T})$.

As building the compiler in the first place (before commencing work on it) would take infeasibly long, there is a `Holmake --fast` option to cheat proofs – this is not a good idea for untested theories, but fine for building a local copy of the verified compiler.

B.3 The goalstack and interactive proof

HOL has interactive plugins for the Vim and Emacs editors so the user can write code in a script file while simultaneously sending it over to a running HOL. This is the best of both worlds: code is evaluated line by line and so easily modified (as in pure REPL), but a persistent store of work also exists (as in pure script). Familiarity with this tool therefore greatly simplified the project.

This allows for interactive, goal-oriented proof – a top-down strategy which starts with the desired result (the goal), and breaks it down into smaller, more tractable subgoals. This process constructs a proof tree, with the choices made in how to decompose the goal forming the proof strategy. HOL automatically generates the *validation*: the proof that all subgoals taken together imply the original goal. The user therefore need only prove each subgoal, and HOL will do the rest.

Goals are managed using the goalstack, an SML structure of records of `(goal list, validation)` pairs: each pair has a list of subgoals and a validation function which takes in a list of theorems (i.e. the resulting theorems from proving the subgoals) and returns a theorem, effectively reversing the splitting of the goal into subgoals. The goalstack is managed through the SML structure, and interactive keyboard bindings aid this – management of the goalstack was another key feature to master for this project.

The proof tools for top-down, goal-oriented proof are tactics and tacticals – these are described below.

B.3.1 Tactics

Tactics are SML functions taking in a goal and returning a list of subgoals and a validation function. `(tactic : goal -> goal list * validation)`. A tactic “solves” a goal if it reduces it to the empty list of subgoals. Tactics are derived from the trusted module of primitive inferences, so they cannot derive false proofs – at worst they can produce unsolvable subgoals, or introduce too many assumptions. This holds even when a tactic is poorly implemented in the underlying SML, making them resistant to failure.

The more common types of tactics (and therefore the main ones used in this project) are summarised below – there are of course many more, but this gives a rough guideline to the basic tools available.

Rewrites. These substitute terms in the goal using equality theorems. The most elegant proofs are purely rewrites, using the transitivity of equality to prove a goal. There are many subtly different types of rewrite, all of which must be understood in order to prove effectively.

Provers. HOL has some automatic decision procedures to find a proof. For example there are natural number solvers, first-order provers, and SAT solvers. Theorems can be supplied as further resolution candidates.

Induction. These start a proof by induction on a universally-quantified goal or recursively-defined data type. For example, inducting over a list breaks the goal into the base case (the empty list, `[]`) and the inductive case (the list constructor, `h::t`) with an inductive hypothesis as an assumption (i.e. assumption on `t`). Using induction theorems generated on definition of a recursive function or data type is of particular use.

Cases. Performing case splits on terms can be useful – this can mirror many compiler functions, which perform a case analysis too.

Instantiation. Instantiating universally- or existentially-quantified variables is a useful tool – for example, to solve an existentially-quantified goal we can provide a *witness*, a term for which the goal holds true. There are a number of ways to instantiate, with varying degrees of automation.

Lemma introduction. Assumptions can be introduced in a great number of ways, with varying degrees of automation. This is usually paired with instantiation, to introduce and instantiate a previously-proved theorem. These were some of the more difficult methods to master.

Resolution. Resolving the goal with its assumptions or a given theorem can create assumptions with useful instantiations. Care must be taken, as the number of assumptions can explode.

Cheat. The `cheat` tactic proves any goal given[3]. This is useful in determining proof strategy: a large goal requires many lemmas in its proof, so these can be cheated to allow progress, and the lemmas can be proved later. This ensures that time is not wasted proving lemmas that will not be useful for the overall goal, and mirrors the top-down approach to proof in general.

B.3.2 Tacticals

The tacticals are in general functions that return tactics. They allow the user to manipulate tactics, and in particular organise proofs – a summary of the more common tacticals is given below.

Tactic composition and sequencing. Tactics can be chained using infix operators. Two tactics can be combined into one which encapsulates sequential execution of the originals (`THEN : tactic -> tactic -> tactic`, often written as `>>`). Subgoals must

[3]This cannot give any false results, as generated theorems are tagged as having been cheated.

be managed carefully – a tactic can generate many subgoals, each of which may need to be treated differently. `THEN` applies both tactics to all subgoals – tacticals such as `THEN1` (or `>-`) are more specific (`THEN1` applies the left-hand tactic, then solves the first subgoal fully with the right-hand tactic). Effective use of these is necessary for creating theories, as they can greatly condense proofs.

Assumption management. Many tacticals exist for selecting assumptions, with type `(thm -> tactic -> tactic)`: they select an assumption, and apply the provided `(thm -> tactic)` function to return a tactic. A common use is to select an assumption and instantiate some of its quantified variables.

B.3.3 Conversions and rules

Bottom-up proof is possible in HOL using conversions and rules, but it is difficult to sequence and combine branches of the proof tree. Conversions map terms to theorems (`conv : term -> thm`), and rules map theorems to other theorems. A conversion can be as simple as mapping `(t : term)` to ($\vdash$ `t` $=$ `t : thm`). However for this project, conversions can be useful to instantiate universally-quantified variables in a general proof, to specialise to a more specific case. The more general case is easier to prove due to the more general assumptions, but the specialised case is more pertinent to the compiler algorithm. This project uses rules to prove properties about the abstract reachability functions (§ *3.3.3*).

B.4 Summary of HOL workflow

The general pattern for HOL theory construction is therefore:

1. Interactively import necessary theories.
2. Interactively set up definitions and types, and state desired theorems in a script file.
3. Interactively prove theorems using tactics and tacticals, operating on the goalstack.
4. Package up the resulting script as a standalone compilation unit (this is required to be an SML structure).
5. Run `Holmake` on the script file to resolve dependencies, and compile and execute the script to return a standalone, reusable theory.

Familiarity with this workflow, in particular with interactivity in Vim, was a necessity for this project. This was achieved with the help of Anthony Fox and Magnus Myreen – the limited documentation and plethora of available tools in HOL make it difficult to approach for a beginner without an experienced guide. Gaining experience through online tutorials, observation of other proof scripts, supervision with Anthony and Magnus, and trial and error was necessary to build a repertoire of available tools. This process carried on throughout the project, and many new tools were discovered even towards its end.

Appendix C

Results of code reduction evaluation in FlatLang

This appendix contains the raw data obtained from measures of code reduction for the optimisation in FlatLang (*Figure C.1*). Eight example programs were used to illustrate code reduction: implementations of six UNIX tools (`cat`, `diff`, `echo`, `grep`, `patch`, and `sort`), a simple "hello world" program, and an implementation of a word count for files. These implementations are provided as part of the CakeML codebase in the `cakeml/examples/` directory. Each of these was compiled to FlatLang, and then optimised using the dead code elimination pass written in this project. Five measures of code size were then use to evaluate the extent to which the optimisation reduced the amount of code. These measures are as follows:

- **Number of top-level declarations.** Each declaration in FlatLang is of type `dec`, and FlatLang code is a `dec list` (§ *A.2.1*). This measure is effectively the size of that list.

- **Expression size.** Provided in the CakeML codebase are measures of expression size for FlatLang expressions – this is the number of constructors required to define the expression. Expressions can be contained within the value declarations over which the optimisation acts (§ *A.2.1*).

- **Number of global variable initialisations.** The number of `(GlobalVarInit : Op)` operations in the code, each of which initialises a global variable. This is therefore the number of global variables that the code analysis determined to be reachable (§ *3.2*).

- **Number of global variable lookups.** The number of `(GlobalVarLookup : Op)` operations in the code, each looking up the value of a global variable.

- **Term size.** This is an SML measure rather than a HOL-implemented one. It represents the size of a HOL term in terms of the number of SML constructors required to express it.

The full results can be found in *Figure C.1.* The data clearly shows significant code reductions for all example programs across all measures.

Program	No. of top-level declarations (each of type `dec`)			Expression size			No. of global variable initialisations			No. of global variable lookups			Term size		
	Before	*After*	*% removed*	*Before*	*After*	*% removed*	*Before*	*After*	*% removed*	*Before*	*After*	*% removed*	*Before*	*After*	*% removed*
`cat`	380	56	85%	10509369	1494879	86%	371	47	87%	407	91	78%	489167	46907	90%
`diff`	396	94	76%	11825222	3353015	72%	387	85	78%	442	145	67%	527386	110565	79%
`echo`	377	47	88%	10416268	1108222	89%	368	38	90%	403	64	84%	486445	37753	92%
`grep`	479	186	61%	21247315	13477031	37%	461	168	64%	571	275	52%	746438	362713	51%
`patch`	389	86	78%	11979935	3642355	70%	380	77	80%	425	13	97%	529131	120635	77%
`sort`	382	74	81%	10921885	2454560	78%	373	65	83%	424	129	70%	504034	82715	84%
Hello world	377	33	91%	10383326	634304	94%	368	24	93%	399	32	92%	485769	20784	96%
Word count	379	76	80%	10508107	2233183	79%	370	67	82%	414	125	70%	489148	80305	84%

Figure C.1: *Raw results of code reduction evaluation of the dead code elimination optimisation in* FLATLANG. *Five measures of code size were evaluated on the code both before and after the optimisation was applied, to give a percentage of code removed. This was determined for eight example programs, which can be found in the CakeML codebase in the* `cakeml/examples/` *directory. In all measures, the optimisation pass shows significant code reduction for all example programs, demonstrating the efficacy and usefulness of the pass.*

Appendix D

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Implementing and verifying a compiler optimisation for CakeML

H. R. Kanabar, King's College
Originator: Dr. Magnus Myreen
16 October 2017

Project Supervisors: Dr. Stephen Kell & Dr. Anthony Fox
Project Advisors: Dr. Magnus Myreen & Dr. Ramana Kumar
Director of Studies: Dr. Timothy Griffin
Project Overseers: Dr. Markus Kuhn & Dr. Peter Sewell

Introduction

This project tackles implementing and verifying global dead code elimination for CakeML, an open-source functional programming language which is a significant subset of Standard ML. Its semantics and compiler algorithm are specified in higher-order logic, and verified with the interactive theorem prover HOL4.

Dead code elimination is a compiler optimisation involving the removal of code which does not affect the results of the program (known as *dead code*). This means that code which will never be executed due to the control flow of the program (known as *unreachable code*), or which only affects *dead variables* (variables that will not be read again), is not compiled. The optimisation can reduce the size of compiled binaries, preventing waste of disk space and instruction memory, and can speed up program execution, as instructions that do not affect observable program behaviour are not executed. Dead code elimination may also enable further optimisations by simplifying program structure. The CakeML compilation pipeline transforms source code to machine instructions via

successive intermediate languages over several *compilation passes* – elimination of dead code should occur as early as possible in this pipeline, to avoid the cost of translating it between the intermediate languages.

As it currently stands, large libraries of CakeML code (known as the basis libraries, as in Standard ML) are prepended onto any user-written source code before compilation. This makes the compiler very slow for even the simplest of programs due to the large amount of code included in these libraries, a large proportion of which may not be used by a given program. This issue is thus the motivation for the project[1].

Once this optimisation has been implemented, its soundness must then be proved – the optimised and non-optimised compilers must produce semantically equivalent code. This verification will be carried out using higher-order logic and the interactive theorem prover HOL4, in the same way that the rest of the language has been proven to be semantics-perserving. The proof of the optimisation must then be integrated into the existing proofs for the compiler as a whole.

Starting point

The CakeML open-source GitHub repository[2] contains all of the CakeML code. This includes the specification and semantics of the language, as well as the compiler algorithm, all specified and verified in higher-order logic using the HOL4 interactive theorem prover. The compiler algorithm transforms the code from source text to machine code, passing through twelve successive intermediate languages along the way, and targeting five different architectures. The compiler has been proven to be correct – it can be shown that it transforms CakeML programs into semantically equivalent machine code. CakeML is in its third main version, and has a fully implemented and verified compiler for its language specification. Many compiler optimisations have also already been implemented and verified, and the codebase comes with a library of regression tests to ensure that new changes are compatible with all prior work. However, the compiler currently prepends the entire basis library onto all source code it compiles – this is therefore open to optimisation.

CakeML uses HOL4 to verify its compiler algorithm. HOL is an interactive theorem prover for higher-order logic[3], and HOL4 is its most up-to-date version, with an active community. Currently the entire compiler algorithm has been verified to be correct using HOL4, so only the work on the new optimisation will require verification.

Resources required

This project will mainly be carried out on my own quad-core (Intel i7-2720QM) laptop, with 8 GB of RAM and 1 TB of hard-drive, running Windows 10 with an Ubuntu 16.04 LTS dual-boot. Version control will be through GitHub, with automatic synchronisation

[1] `https://github.com/CakeML/cakeml/issues/337`
[2] `https://github.com/CakeML/cakeml`
[3] `https://hol-theorem-prover.org/`

to Google Drive and regular (at least weekly) backups to my own external hard-drive (1 TB).

In addition, the project will require extensive outside assistance from Ramana Kumar and Magnus Myreen, two members of the CakeML development team – this will mostly be by email and Slack. Anthony Fox has also agreed to help with HOL setup through pair-programming and supervision, which will require regular access to the Department of Computer Science and Technology. Stephen Kell will therefore be acting as a contact point and proxy supervisor for the project, with Ramana, Magnus, and Anthony providing the majority of the technical expertise. Stephen will be on hand to help mostly with project planning and dissertation work.

Work to be done

The overall project can be further broken down into the following sub-projects:

1. **Starter Project:** dead code elimination in wordLang, an intermediate language in the middle of the compilation pipeline.
   (a) Implement code reachability analysis functions in HOL for wordLang.
   (b) Verify the correctness of the abstract reachability function.
   (c) *(time-permitting)* Prove that the deletion of unreachable wordLang functions preserves semantics.
2. **Main project:** dead code elimination in flatLang, an intermediate language early on in the compilation pipeline.
   (a) Implement a bottom-up compiler pass for deleting unreachable declarations in a CakeML program (source programs in CakeML are essentially lists of declarations).
   (b) Verify that the observable semantics is preserved by the new compiler pass.
   (c) Integrate the new compiler passes (wordLang and flatLang) into the main CakeML repository.

Success citeria

The project will be determined to be successful if the following criteria are achieved:

- A global dead code elimination pass is implemented for at least flatLang.
- The implementation of the pass fits into and is part of the latest version of the CakeML compiler.
- The new compiler pass for wordLang has been proved correct, and its correctness theorem has been used to update the overall theorem for the existing compiler.
- Simple test cases demonstrate that the new compiler pass is able to remove unused declarations from the basis library and from user-written code.

Possible extensions

If the main aims are achieved, the following extensions will be considered:

1. **Verify the optimisation for flatLang.** The new compiler pass for flatLang should be proved correct, and its correctness theorem then used to update the overall theorem for the existing compiler.

2. **Empirically determine the efficiency of the new optimisation(s).** The CakeML development team estimate that both binary size and compilation times can be reduced by 50% for simple examples that use little of the lengthy basis libary, if a good global dead code elimination pass is correctly implemented.

3. **Verify other optimisation passes for flatLang or wordLang.** In flatLang, a possible optimisation is a global purity analysis pass – this will move pure constant computations out of loops, turning them into new declarations with fresh names.

Timetable

Planned starting date is 19/10/2017.

1. **Michaelmas weeks 3–4:** Learn the basics of HOL4, with help from Anthony Fox and online tutorials. Familiarise with the theory of dead code elimination, and the CakeML compiler specification.

2. **Michaelmas weeks 5–6:** Create basic test examples and performance metrics. Start definitions and proofs of code reachability in wordLang. Implement code reachability analysis functions in wordLang.

3. **Michaelmas weeks 7–8:** Finish proving code reachability function correctness. Prove that deleting unreachable code does not change semantics of wordLang *(time-permitting)*.

4. **Michaelmas vacation:** Integrate with existing CakeML codebase, including running regression tests. Demonstrate efficiency gains made using test examples and performance metrics. Familiarise with flatLang, including discussion of deadcode elimination in flatLang with the CakeML development team.

5. **Lent weeks 0–2:** Write progress report. Begin work on dead code elimination from flatLang – create definitions and proofs in HOL4.

6. **Lent weeks 3–5:** Implement compiler pass for dead code elimination in flatLang. Verify that this preserves semantics.

7. **Lent weeks 6–8:** Run regression tests from CakeML codebase. Update existing compiler correctness theorem to take into account the new optimisation pass.

8. **Easter vacation:** Finish integration with CakeML codebase. Work on extension projects *(time-permitting)*. Write main dissertation chapters.

9. **Easter weeks 0–2:** Use test cases and examples to illustrate performance gains. If any extension is complete, generate test cases for this *(time-permitting)*. Finish drafting dissertation, and submit to supervisor for feedback.

10. **Easter weeks 3–4:** Continue to test, document, and integrate code. Proof-read dissertation, then submit to supervisor. Make corrections, and make final submission.