Y. K. Tan

Verified Register Allocation for CakeML

Computer Science Tripos – Part II

Robinson College

May 8, 2015

Proforma

Name / College:	**Yong Kiam Tan / Robinson College**
Project Title:	**Verified Register Allocation for CakeML**
Examination:	**Computer Science Tripos – Part II, June 2015**
Approximate Word Count[1]:	**11998**
Project Originators:	**Dr M. Myreen / R. Kumar**
Project Supervisor:	**R. Kumar**

Original Aims of the Project

My project's overarching aim was to implement and verify a register allocation pass for the verified CakeML compiler backend. This was split into two major steps:

1. Liveness analysis for an intermediate language of CakeML's compiler backend.
2. Simple register allocation using a graph colouring framework.

Both steps should be verified in HOL4 and a final theorem that links up the entire pass should be produced. The theorems should also be suitable for the entire pass to fit into the overall correctness proofs of the CakeML compiler.

Work Completed

All the initial aims of this project have been successfully achieved. My method of implementation allowed me to verify a significantly more complex register allocation algorithm than expected in the original proposal. Based on quantitative tests, it achieves roughly 1.3x coalescing performance over the baseline algorithm.

Register allocation performance is generally improved by placing code into Static Single Assignment form. As an extension, I also implemented and fully verified a pass that performs this transformation.

Special Difficulties

None.

[1]This word count was computed using `texcount -sub=chapter diss.tex`.

Declaration

I, Yong Kiam Tan of Robinson 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

Date

Contents

Acknowledgements

This project would not have been successful without the guidance of my project supervisors, Ramana Kumar and Dr. Magnus Myreen. I would also like to thank Prof. Mike Gordon, Prof. Alan Mycroft, Dr. Alastair Beresford, Matthew Chua, Vincent Huang and Daniel Low for useful advice and discussions over the course of this project.

Chapter 1

Introduction

This dissertation gives an overview of the implementation and verification of a register allocation pass for the verified CakeML compiler. In this chapter, I introduce the CakeML compiler (§1.1, §1.2) and discuss the need for a register allocator (§1.3). The remaining sections describe the project (§1.4, §1.5) and explain the conventions used in this dissertation (§1.6).

1.1 CakeML

CakeML [13] is a significant subset of Standard ML (SML) with a compiler implemented in the HOL4 Theorem Prover [4]. The CakeML compiler is fully **verified** i.e. we have a correctness theorem that links the behaviour of (well-typed) input CakeML programs to the behaviour of the corresponding x86 assembly produced by the compiler.

Compilers are a significant part of our trusted computing base; we trust that the compiler correctly transforms input programs into machine code. However, no amount of verification done on user programs can prevent bugs introduced by the compiler. Along more malicious lines, Ken Thompson discussed the problem of trusting "program-handling programs" (e.g. compilers) in his Turing Award lecture [17].

Fully verifying the compilation process partially closes these loopholes in software verification: we can trust the compiler at least as much as we trust the theorem prover's logical core.

1.2 Compiler Organisation

As with most compilers, the compilation process from high-level CakeML source code down to machine code is broken into a series of **transformation** steps. Each transformation corresponds to a translation towards progressively simpler intermediate languages (ILs). This gives us a natural way to split up the verification process: we first verify the correctness of each translation step before composing these individual correctness theorems into a final theorem for the entire compilation process. Figure 1.1 shows an example overview; every solid edge is a translation step with an associated correctness theorem.

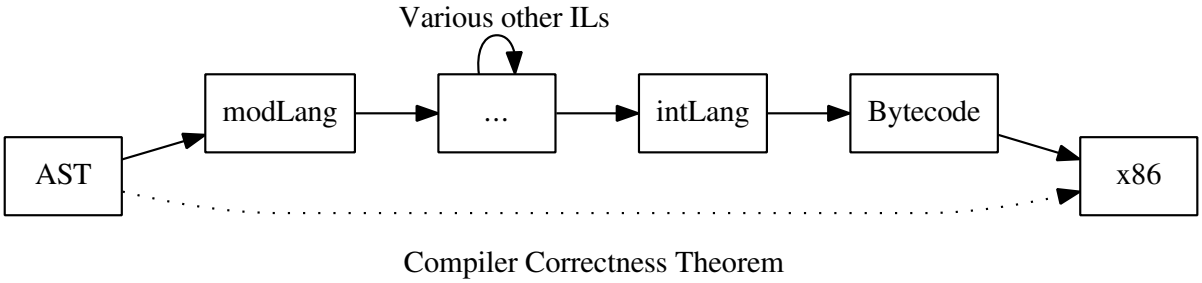

Figure 1.1: A part of CakeML's current backend.

To support reasoning about individual transformations, CakeML uses formally defined big-step style evaluation semantics for each IL. The correctness theorem for each translation step shows that the translation preserves the semantics[1] of the program before and after translation. A verified implementation of the compiler as described above is available [3].

To increase the compiler's usability, a new backend has been designed to perform more powerful compiler optimisations and target more instruction sets. The register allocator is a key component of this design as it is needed to achieve both aims.

1.3 Register Allocation

During the code transformation process, compilers usually assume that there are an unlimited number of temporary locations available to store generated temporaries; an example of temporaries being generated is shown in Table 1.1.

Input Expressions	Output 3-Register Code
	$t_1 := a + b$
$x := a + b + c + d$	$t_2 := c + d$
$y := (a + b) * e$	$x := t_1 + t_2$
	$y := t_1 * e$

Table 1.1: Expression Flattening.

As the compiler moves towards machine code, it eventually needs to start targeting a specific Instruction Set Architecture (ISA). Modern ISAs typically provide a small number of registers on which we can perform computations. **Register Allocation** is the task of assigning the large number of generated temporary locations to this fixed, finite set of registers.

The following list shows important register allocation objectives considered in this project. Note that the first two objectives are compulsory objectives (requiring proof) while the latter three are important optimisation goals.

[1]More precisely, it is proved that a correctness invariant holds between the semantics of the input program and the semantics of the output program. The exact form of these invariants depends on the ILs and the transformations being applied.

1. Register allocation must be a **safe** transformation i.e. the input and output code should behave the same way with respect to the semantics.

2. The resulting code must **respect conventions** required by subsequent compilation steps. For example, we may require that function calls always place their return values into a fixed register r_0. The register allocator must ensure that these conventional registers are appropriately assigned.

3. The allocation should seek to minimise the number of registers used in the output code. Ideally, the number used should be at most the **number of registers provided by the target ISA**, k. Note that k is used throughout this dissertation to refer to this value[2].

4. In the cases where k is insufficient, we need to minimise the costs from **spilled** variables. Spilling a variable means keeping it on the stack; extra load/store instructions are required whenever we need to do computations on a spilled variable. Having a large number of spilled variables is clearly undesirable.

5. A **coalesced** move instruction is one where both operands are allocated to the same register. The resulting instruction can be deleted to yield more efficient output code. Coalescing is an important optimisation goal because we often need to move/copy variables around during compilation. Table 1.2 illustrates the use of coalescing; note also that the allocator is forced to always assign t_0 to r_0 in order to satisfy the return value conventions.

Input Code	Calling Conventions	Register Allocation	Eliminate Redundancy
$x := a + b$ `return` x	$x := a + b$ $t_0 := x$ `return` t_0	$r_0 := r_1 + r_2$ $r_0 := r_0$ `return` r_0	$r_0 := r_1 + r_2$ `return` r_0

Table 1.2: Setting up calling conventions and coalescing moves.

Unsurprisingly, these objectives together form an NP-Hard optimisation problem [8] so register allocators are typically heuristic in nature. A common technique, which is the one used in my project, is to view the task as a graph colouring problem. The next chapter explains this isomorphism and the various algorithms used to achieve the desired objectives.

1.4 Project Description

My primary project goal was to verify a register allocator for the new CakeML compiler backend. This is supported by a backend IL, **wordLang**, on which code transformations are performed. Additionally, three extensions identified in the proposal were completed:

[2]The value of k varies across different ISAs so it is given as an input to the allocator.

1. The Iterated Register Coalescing (IRC) register allocation algorithm was implemented.

2. An additional pass was written (and verified) to make input code more suitable for register allocation.

3. The allocator was successfully run through CakeML's bootstrapping mechanism.

All algorithms used in this project are fairly standard and well known. Therefore, the main project difficulty, and focus of this dissertation, is the formal verification of their implementations.

1.5 Related Work

I initially proposed to extend David Barker's Part III Project [6] where he verified a register allocation framework based on Chaitin's register allocator [11] together with some heuristics proposed by Briggs [9, 10]. His project was an important source of ideas but extending it was abandoned for two reasons:

1. His framework is verified for straight line (non-branching) 3-register code but wordLang contains more complicated control flow and program constructs, so direct extension was not entirely feasible.

2. An alternative method of proof used in my project allowed me to implement the more complicated IRC algorithm proposed by George and Appel [12].

My project also builds on substantial prior experience and work with CakeML. In particular, I was already familiar with theorem proving in HOL4, the CakeML compiler and the design of its new backend (e.g. the semantics of wordLang).

1.6 Formatting Conventions

Throughout this dissertation, I use a mix of pseudocode, informal and formal definitions. The conventions adopted are as follows:

- Code transformations and pseudocode examples are presented in tables (e.g. Table 1.1).

- Informally defined terms are in **bold**. Their intended definition should be clear from the surrounding context.

- Formally defined terms and theorems are printed directly from HOL4. They appear in a *script* or `typewriter` font and may also contain standard logic symbols.

- Every Lemma/Theorem is followed by an outline of its proof in HOL4. The low level proof details (e.g. tactics and trivial lemmas) are omitted.

Chapter 2

Preparation

This chapter covers the necessary background for this project. It begins with brief overviews of Interactive Theorem Proving (§2.1) and the semantics of wordLang (§2.2). Next, the standard terminology and algorithms for register allocation are defined (§2.3, §2.4). The final section discusses the requirements of this project (§2.5).

2.1 Interactive Theorem Proving

HOL4 is an **interactive theorem prover** i.e. proofs are done by the user with assistance from the theorem prover. The CakeML compiler is defined as terms in the HOL4 logic and we verify its correctness by writing proofs about these terms.

2.1.1 Tactic Proof

A proof is started by stating a **goal**, i.e. a proposition that we want to prove. We then apply **tactics** to the current goal to change it into a list of zero or more subgoals. Subsequent tactics are applied onto these subgoals and the proof is completed when there are no subgoals left. This backwards style of proof is commonly known as **tactic proof** and is similar to what one might use in a sequent calculus. HOL4 provides a large number of tactics to interact with the proof manager, for example:

- `REWRITE_TAC`: Rewrite the goal with a list of definitions and theorems.
- `FULL_SIMP_TAC`: Recursively simplify the assumptions and goals.
- `METIS_TAC`: Automatic first order theorem prover.

2.1.2 Induction and Proof Generalisation

Induction is a fundamental tool for reasoning about recursively defined semantics and functions. For example, to prove a property about an algorithm over lists, we can apply an induction tactic on the following list induction theorem to break our goal down to the base case, `[]` and the inductive case, $h::t$.

$$\vdash \forall P.\ P\ [] \wedge (\forall t.\ P\ t \Rightarrow \forall h.\ P\ (h::t)) \Rightarrow \forall l.\ P\ l$$

To successfully do a proof by induction, it is important to find a suitable **generalisation** of our final goal theorem. As a motivating example, suppose we have the following algorithm that computes the length of a list using an accumulator:

```
list_length_acc [] n = n
list_length_acc (x::xs) n = list_length_acc xs (n + 1)
```

We want to prove a correctness theorem of the form:

$\vdash \forall ls.$ `list_length_acc` ls `0 = LENGTH` ls

Instead of proving this directly, we instead prove a generalised version by induction (notice that setting n `= 0` yields what we needed):

$\vdash \forall ls\ n.$ `list_length_acc` $ls\ n$ `=` n `+ LENGTH` ls

Much of the verification effort for this project went into finding suitable generalisations of the relevant theorems for induction to work.

2.1.3 Proof Composition

When dealing with a complex goal (e.g. correctness of the CakeML compiler), it is often difficult to attempt a proof directly. Proof complexity can be dealt with by **decoupling** (or decomposing) the proof into smaller correctness lemmas (e.g. correctness of individual compilation steps) before **composing** these lemmas to prove the full theorem.

For example, if we separately prove the premises of the following rule, then we can immediately derive the composed correctness theorem in its conclusion:

$$\frac{\begin{array}{c}\forall prog.\quad P\ prog \Rightarrow R\ (trans_1\ prog) \\ \forall prog.\quad R\ prog \Rightarrow Q\ (trans_2\ prog)\end{array}}{\forall prog.\quad P\ prog \Rightarrow Q\ ((trans_2 \circ trans_1)\ prog)}$$

This is similar to the Sequence rule in Hoare Logic; it is important to find an appropriate predicate R that **links** the transformations. In general, more complex decoupling and composition techniques may be required.

2.2 wordLang

The wordLang IL is designed to support **variable renaming transformations**: these take an input wordLang program, rename some variables and output a new wordLang program. The register allocator is treated as one such transformation, i.e. the input and output of the allocator will be wordLang programs. Therefore, it is crucial to understand the semantics of wordLang in order to verify the correctness of the register allocator.

2.2.1 wordLang Definition

The wordLang syntax is defined using a grammar encoded as a data structure in HOL4. This yields wordLang programs of type (*prog* :α `word_prog`). From the syntax, we define the semantics using an evaluation function of type:

```
(wEval :α word_prog × α word_state ->
        α word_result option × α word_state)
```

`wEval` takes as input a wordLang program (and state) and gives the evaluation wordLang result (and result state). Definition 2.2.1 shows a simple clause in its definition.

Definition 2.2.1. `wEval` (`Move`)

```
wEval (Move pri moves,s) =
if ALL_DISTINCT (MAP FST moves) then
  case get_vars (MAP SND moves) s of
    NONE ⇒ (SOME Error,s)
  | SOME vs ⇒ (NONE,set_vars (MAP FST moves) vs s)
else (SOME Error,s)
```

The full AST and evaluation semantics of wordLang is lengthy so its full definition is given in Appendix A. Aside from a few high-level commands (e.g. exceptions and heap allocation), wordLang largely resembles standard 3-register code. The rest of this section highlights important features in its definition.

2.2.2 Control Flow

There are two types of control flow expressible in wordLang programs, `Seq` commands give sequential control flow, while `If` and `Call` commands provide forward branching control flow[1]. In particular, the only way to achieve looping is via function recursion. A **wordLang program** should be thought of as the body of a function (that contains no loops) and we will be performing register allocation on these function bodies.

Evaluation results have option type: `NONE` results indicate that the program should continue evaluation while `SOME` results indicate that evaluation has finished or there was an error. This is illustrated in Definition 2.2.2 for the semantics of `Seq`, where we only continue evaluating c_2 if c_1 evaluates to `NONE`.

Definition 2.2.2. `wEval` (`Seq`)

```
wEval (Seq c1 c2,s) =
(let (res,s1) = wEval (c1,s)
 in
   if res = NONE then wEval (c2,s1) else (res,s1))
```

This semantics ensures that whenever our result is not `NONE`, then control flow necessarily exits from the current program body i.e. we do not need to continue evaluating the rest of the commands.

[1] A `Call` command might branch to exception handling code.

2.2.3 Locals, Stack and the Garbage Collector

The wordLang stack is essentially a standard call stack with one minor difference: the stack frame of the currently executing function is not thought of as part of the stack. Instead, the currently executing function has an extra state component, the **locals**, modelled as an unordered finite map of variables to values. The rest of the **stack** is a list of stack frames, each containing an ordered list of variable-value pairs for that frame. This distinction is useful because most wordLang commands only interact with local variables; we can reason about their semantics separately from the stack.

The semantics of wordLang also describes an abstract black-box Garbage Collector (GC) function[2]. As with standard GC algorithms (e.g. Mark and Sweep), the abstract GC will need access to the **root set** and heap memory in order to do its task of freeing unreachable objects. The root set is the set of all values reachable from the current execution (i.e. values of the local variables and every value on the stack).

There are two commands that interact with the stack: `Call` jumps to another function and `Alloc` calls the GC before allocating heap space. To make the root set easy to reason about, the semantics of both commands push the current locals into a newly created stack frame so that the GC only needs to access values on the stack.

Creation of a stack frame is done by imposing an ordering on the unordered set of local variables. Definition 2.2.3 shows how this is defined: we first lexicographically sort the locals (`toAList`, followed by `QSORT`) to get an initial order, then permute the resulting list (`list_rearrange`) using a permutation provided by the **permutation oracle**.

Definition 2.2.3. `env_to_list`

```
env_to_list env bij_seq =
(let mover = bij_seq 0 in
 let permute n = bij_seq (n + 1) in
 let l = toAList env in
 let l = QSORT key_val_compare l in
 let l = list_rearrange mover l
 in
   (l,permute))
```

2.2.4 Permutation Oracles and Variable Renaming Transformations

To understand the permutation oracle, consider Figure 2.1 which shows an example where local variables are renamed during execution. Table 2.1 shows how a stack frame is created from these two sets of local variables. There are two points to note about this example:

1. The Original and Renamed stack frames were permuted differently.

2. The values of the resulting permuted stack frames (highlighted in bold) are exactly the same. This is not true of the frames if we had simply sorted them.

[2]The GC will be implemented at a lower level than wordLang so it is abstracted in wordLang.

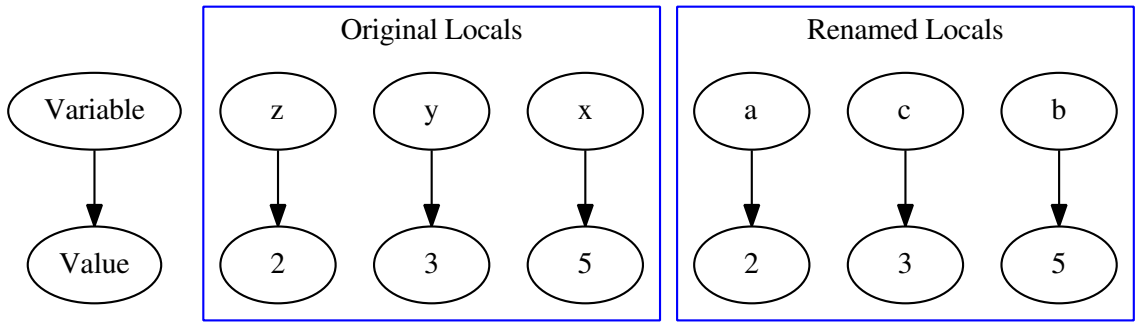

Figure 2.1: Local variables after a variable renaming transformation.

Sorted (Original)		Permuted (Original)		Sorted (Renamed)		Permuted (Renamed)	
x	5	y	**3**	a	2	c	**3**
y	3	x	**5**	b	5	b	**5**
z	2	z	**2**	c	3	a	**2**

Table 2.1: Creating a stack frame using the permutation oracle.

The purpose of the permutation oracle is precisely to guarantee that the values in every stack frame on the stack appears in the same order before and after a variable renaming transformation. With this guarantee, the root sets seen by the GC will be exactly the same, and this forces it to have the same behaviour in both cases. This is used to decouple reasoning about the GC when we do a variable renaming transformation: to show that the GC behaves the same way before and after the transformation, it suffices to prove that the permutation oracles work as expected.

The oracle is formally modelled as an infinite sequence of bijective functions. The head of the oracle is popped and used to permute our stack frame each time it is needed. For every variable renaming transformation in wordLang (e.g. register allocation), we have to construct an appropriate permutation oracle in its proof of correctness. Intuitively, the permutation oracle needs to undo the effects of the variable renaming transformation on stack locations.

2.2.5 Interaction with other ILs

The correctness proofs for wordLang transformations have to be composed properly with the correctness proofs from adjacent ILs. Figure 2.2 shows the position of wordLang in the new compiler backend; we need the (dotted) correctness theorem to be composed from individual correctness theorems.

Correctness Theorem

The language before wordLang, BVP, has no abstract GC; its compilation theorem into wordLang will be proven correct for all permutation oracles. In contrast, one of the languages after wordLang implements a concrete GC. The compilation theorem out of

wordLang will therefore be existentially quantified to be correct for some permutation oracle. The correctness theorem for wordLang transformations needs to account for the asymmetry of these adjacent correctness theorems[3].

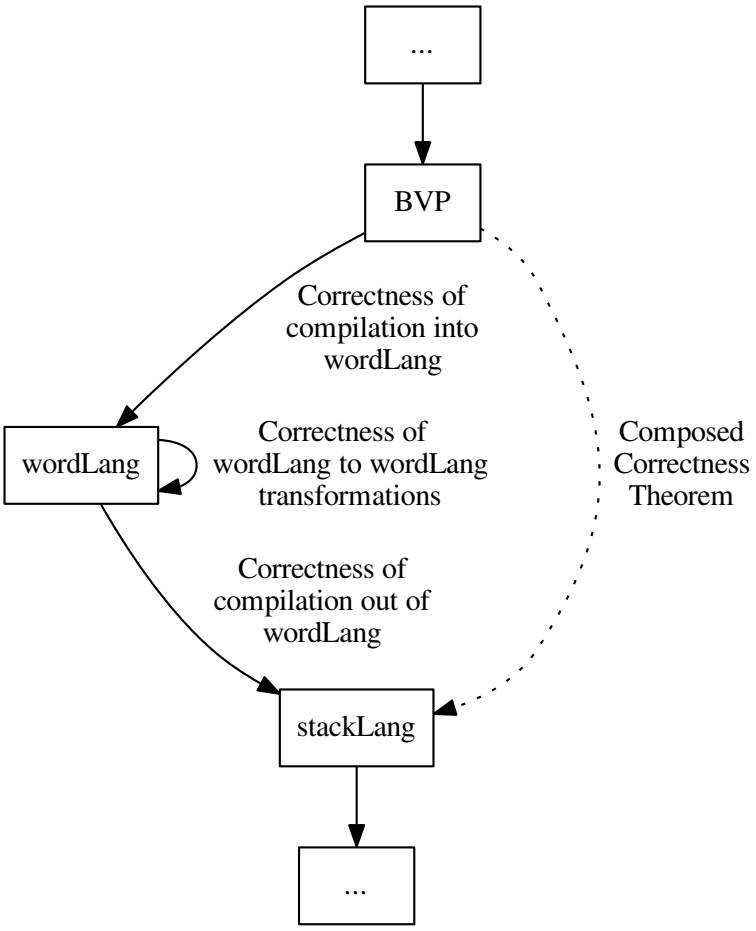

Figure 2.2: Composition of compilation theorems.

Conventions Theorem

The semantics of wordLang itself does not distinguish between different variables names (e.g. registers and stack locations). Instead, we implicitly adopt **impure names** for variables to be interpreted in a later pass of the compiler (e.g. even variables are registers). Later compilation steps will need to have access to a **conventions** theorem about the register allocator in wordLang (e.g. every variable in its output program is even).

[3]The permutation oracle is a key decoupling technique used in CakeML's new backend design; the asymmetry occurs because of this decoupling.

2.3 Liveness Analysis

Liveness Analysis is the key to safe register allocation. The data-flow structure of a program implicitly creates constraints between its variables; some of them cannot simultaneously occupy the same register or stack location. Variables that are constrained in this way are said to **clash** with each other.

To identify clashing variables, we first capture the data-flow in a program using the standard liveness equations. The functions *ref* and *def* extract the reads and writes of a command respectively.

$$\mathit{liveout}(n) = \bigcup_{p \in \mathit{pred}(n)} \mathit{livein}(p) \tag{2.1}$$

$$\mathit{livein}(n) = \mathit{liveout}(n) \setminus \mathit{def}(n) \cup \mathit{ref}(n) \tag{2.2}$$

Combining Equations 2.1 and 2.2 yields Equation 2.3, whose solution defines the **live sets** at each program point. These are statically analysed sets, so they are actually safe over-estimates of the semantically live sets.

$$\mathit{live}(n) = \left(\bigcup_{p \in \mathit{pred}(n)} \mathit{live}(p) \right) \setminus \mathit{def}(n) \cup \mathit{ref}(n) \tag{2.3}$$

All variables contained in the same live set clash with each other but there are additional subtleties:

- Some commands in wordLang can corrupt registers[4]. For example, when we make a function call, the return value is always written back into a local variable even if we do not need it.
- Variables assigned in dead code do not appear in live sets.

In both cases, we encounter a situation where there is a write to a variable that is not live but still clashes with variables in a live set. The latter is fixed by dead code elimination, but a uniform solution is to extend our notion of live sets to **clash sets**, as defined in Equation 2.4.

$$\mathit{clash}(n) = \mathit{liveout}(n) \cup \mathit{def}(n) \tag{2.4}$$

This distinction is demonstrated in Table 2.2 where both subtleties appear in the input code (x cannot be assigned the same register as y or z). We will verify that any allocation of variables to registers that satisfies the constraints generated by clash sets is safe.

2.4 Graph Colouring Algorithms

Given the clash sets at each program point, the register allocator now has to produce an allocation that respects these sets. The first step is to convert the clash sets into a more convenient form.

[4]This is not just a quirk of wordLang, it may also happen in ISAs with non-orthogonal registers.

Input Code	Live Sets	Clash Sets	Register Allocation
	$\{\}$	$\{\}$	
	$x := 5$	$x := 5$	
$x := 5$	$\{x\}$	$\{x\}$	$r_0 := 5$
$y := x + 4$	$y := x + 4$	$y := x + 4$	$r_1 := r_0 + 4$
$z := f(x)$	$\{x\}$	$\{x, y\}$	$r_1 := f(r_0)$
`return` x	$z := f(x)$	$z := f(x)$	`return` r_0
	$\{x\}$	$\{x, z\}$	
	`return` x	`return` x	

Table 2.2: Extracting live and clash sets from input programs.

2.4.1 Isomorphism to Graph Colouring

We define the **clash graph** to be the graph with program variables as its set of vertices and an edge between two vertices if they clash. This can be constructed from a list of clash sets of the program.
A k-colouring is an assignment of values $0, 1, \cdots, k-1$ to vertices of the graph such that no two vertices sharing an edge have the same colour. If we can find a k-colouring, then we equivalently have an assignment of the program variables to registers $r_0, r_1, \cdots, r_{k-1}$.

2.4.2 Simplify and Spill

The basic graph colouring algorithm proceeds by iteratively extracting vertices from the graph onto a **colouring stack**. This stack represents the order in which we will be choosing colours for the vertices. A surprisingly powerful heuristic is the iteration between the following two steps on the input graph:

- **Simplify** removes a vertex of degree $< k$ and places it onto the stack. We defer colouring these **low degree** vertices because we can always colour them regardless of the colours of their neighbours.
- **Spill** occurs when we are unable to Simplify the graph any further. In this case, we can use any reasonable heuristic to pick a vertex to place onto the stack. This is done in the hope that the removal of that vertex allows more opportunities for Simplify.

When all the vertices of the graph have been extracted, the colouring stack is traversed in the **Select** step. Select repeatedly pops vertices off the top of the stack and colours it with a colour that is not already colouring one of its neighbours. The **Biased Selection** heuristic is used whenever there is a choice of colours: we try to pick one that results in coalesced moves for that vertex. If we are unable to find any colour for a vertex, it is marked as a **true spill**.

2.4.3 Coalescing

The next step up in complexity allows us to treat coalescing with more care. A vertex is **move-related** if it is involved in a move and a move is **available** if its operands do not already clash (if they do, we cannot coalesce it).

- **Reckless Coalesce** picks any available move ($u := v$) and merges the two vertices representing u and v in the clash graph. This merge is recorded and we commit to assigning u and v the same colour. This is "reckless" because it might result in more spills.
- **Coalesce** is less reckless in that it only picks moves that satisfy the Briggs or George criterion. The exact form of these criteria can be found in the literature [1]; they both guarantee that coalescing never results in more spills.

Both forms of coalescing will be useful; **Briggs-style coalescing** repeats Coalesce until no further moves can be coalesced before entering the Simplify-Spill loop.

2.4.4 Iterated Register Coalescing (IRC)

The IRC algorithm further refines this idea by moving Coalesce into the Simplify-Spill loop. Simplify may reduce the degree of some vertices, making more moves safely coalesceable. To achieve this, the vertices are partitioned into move-related and non-move-related sets. Simplify now only applies to non-move-related vertices of low degree. An additional step controls whether a vertex is move-related:

- **Freeze** occurs when we have no vertices that can be simplified or coalesced. In that case, we try to pick a low degree, move-related vertex and ignore all the moves it is involved in (treating it as non-move-related).

Figure 2.3 shows the increasingly complex control flow of all the algorithms described. The solid arrows are taken on successful applications of a step and dotted arrows otherwise.

2.4.5 Key Simplifications

All the algorithms described above delay the decision to make true spills until the Select step; this is known as **optimistic spilling**. On a true spill, standard algorithms typically rewrite the input program to handle the spill and then repeat register allocation. In Figure 2.3, there would be an additional dotted edge from Select back to the start in every algorithm. This is needed because the standard algorithms are run with all the registers available in the ISA so there are no spare registers to handle spilled variables.

In the context of wordLang, we make the simplifying assumption that some registers are reserved for movement to-and-from the stack in later stages of the compiler. This is necessary to decouple later compilation passes from wordLang and it also allows us to ignore the rewriting step.

As highlighted in Figure 2.3, this simplification separates out the **Graph Analysis** part of the algorithm from the Select step. Graph Analysis generates the colouring stack

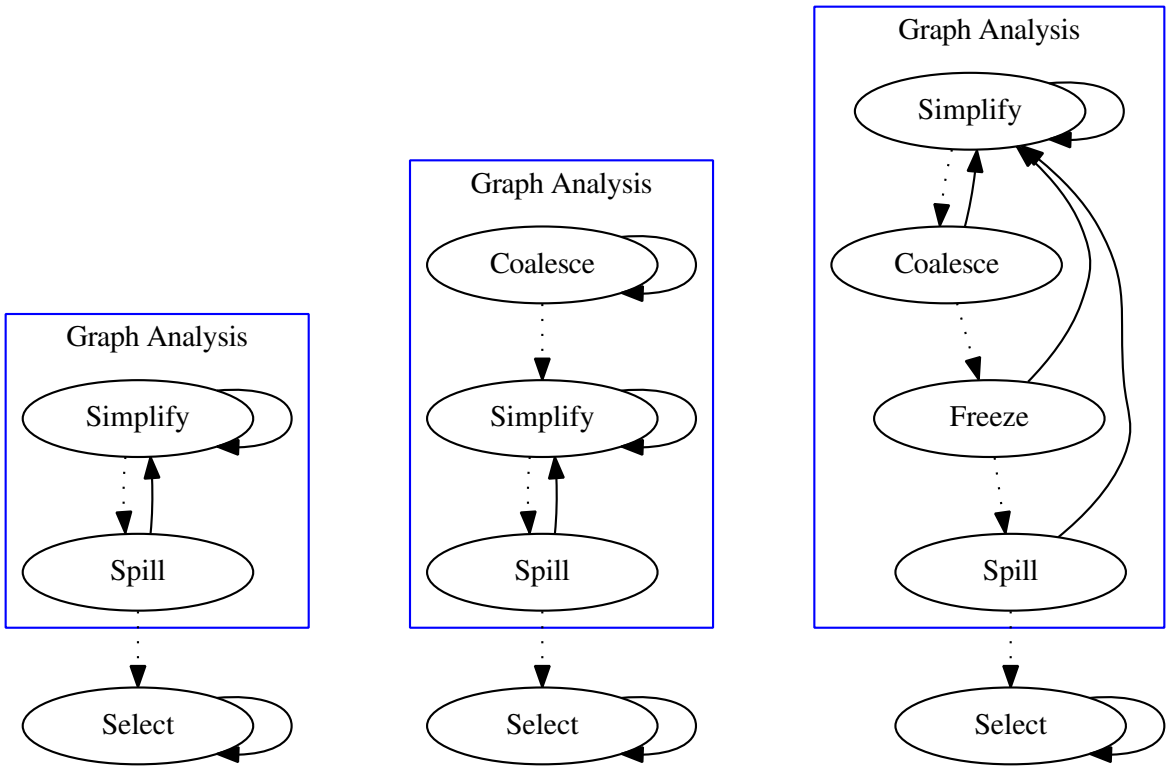

Figure 2.3: From left to right: Simplify-Spill, Briggs-style coalescing, IRC control flow.

while Select is the step that actually colours the graph. I will therefore refer to Select as the **Graph Colouring** step.

2.4.6 Stack Allocation

An additional optimisation step allocates spilled variables to positions on the stack. Performing **Stack Allocation** serves two purposes: it makes the resulting stack smaller, and allows us to coalesce moves between spilled variables. The algorithm used for stack allocation is to first perform repeated Reckless Coalescing; this is safe because we are not bound by the number of registers. This is followed by a Simplify-Select iteration. The implementation and verification details are very similar to register allocation so I will omit further discussion of this step.

2.5 Requirements Analysis

The discussion so far shows that register allocation can be naturally partitioned into three steps: Liveness Analysis, Graph Colouring and Graph Analysis. Successfully implementing and verifying each of these steps is the basic requirement for this project. The additional requirements imposed by a verification project can be divided into engineering and compatibility requirements.

2.5.1 Proof Engineering

Proper engineering is just as important to writing a proof as writing a piece of software; familiar concepts from software engineering can be applied to proofs as well.

Modularity

Even though we can neatly partition the allocator implementation into three steps, it is not immediately obvious that the overall correctness proofs can be partitioned in the same way. As we have seen in §2.1, we need to find a corresponding decoupling of the proofs for each step so that the individual theorems produced can be successfully composed to yield the desired final theorems.

Proof Reuse

Both HOL4 and CakeML repositories already contain a large number of theorems and definitions. It is prudent to re-use these theorems wherever possible instead of spending effort re-proving them. For example, I made heavy use of HOL4's list and functional array datatypes when defining the register allocator. Correspondingly, their associated correctness theorems were used in my proofs.

Iterative Development

The three increasingly complex register allocators discussed in §2.4 follow a similar general pattern. This allowed me to **incrementally** implement and verify the three algorithms; it was much easier to build up the proof of the IRC algorithm than to write it from scratch. Verifying a simpler allocator first also allowed me to check that the theorems produced compose in the expected way before improving the allocator itself.

2.5.2 Proof Compatibility

It is important to ensure that the theorems produced for this project can be used in the overall correctness proof of CakeML's new backend.

Proof Composition

As discussed in §2.2, we need to produce theorems that can be composed with the correctness theorems of adjacent ILs. Additionally, the correctness theorems for register allocation should easily compose with those of other wordLang passes. This is important as it allows us to verify other transformations in wordLang.

Software Tools

My project is entirely implemented and verified using HOL4 (running in Poly/ML). All development was done on a branch of the CakeML compiler hosted on Github.

Chapter 3

Implementation

This chapter describes the implementation and verification work undertaken in this project. It begins with an overview of the proof decoupling (§3.1). This is followed by the implementation and main correctness theorems for Liveness Analysis (§3.2), Graph Colouring (§3.3) and Graph Analysis (§3.4). The entire register allocation pass is defined and verified in (§3.5). Finally, an additional pass and its proof are explained (§3.6). For brevity, long definitions have been trimmed and marked with **abridged**.

3.1 Proof Decoupling Overview

Following the discussion of §2.5, we shall decouple our correctness proofs to fit the three register allocation steps. Figure 3.1 gives a graphical overview; each solid edge is accompanied by its associated correctness theorems and the red circles show where decoupling/linking is used.

3.1.1 Preliminary Definitions

The underlying abstraction across all three steps is a colouring function that colours the variables (or vertices). Since variables in wordLang have type `num`, these functions have corresponding type (*f* `:num -> num`).

Given a colouring function, we can apply it recursively to a wordLang program using `apply_colour` (Definition 3.1.1). This yields a **coloured program**.

Definition 3.1.1. `apply_colour` (abridged)

```
apply_colour f (Move pri ls) =
Move pri (ZIP (MAP (f ∘ FST) ls,MAP (f ∘ SND) ls))
apply_colour f (Seq s1 s2) =
Seq (apply_colour f s1) (apply_colour f s2)
apply_colour f (If cmp r1 ri e2 e3) =
If cmp (f r1) (apply_colour_imm f ri) (apply_colour f e2)
  (apply_colour f e3)
```

To describe the desired conventions theorems, we use two top-level predicates: `pre_alloc_conventions` and `post_alloc_conventions` (Definitions 3.1.2 and 3.1.3).

We expect input code to the allocator to satisfy the former and the corresponding output code should satisfy the latter. Subsequent compilation steps will assume that `post_alloc_conventions` holds of the program. These predicates impose impure interpretations onto the variable names, e.g. the condition `every_var is_phy_var` *prog* corresponds to the constraint that every variable has been allocated to a register (or stack location).

Definition 3.1.2. `pre_alloc_conventions`

`pre_alloc_conventions` *prog* $\iff$
`every_stack_var is_stack_var` *prog* $\wedge$ `call_arg_convention` *prog*

Definition 3.1.3. `post_alloc_conventions`

`post_alloc_conventions` k *prog* $\iff$
`every_var is_phy_var` *prog* $\wedge$ `every_stack_var` $(\lambda x.\ x \geq 2 \times k)$ *prog* $\wedge$
`call_arg_convention` *prog*

3.1.2 Decoupling Liveness Analysis

Liveness Analysis is the only step that interacts with wordLang semantics; the other steps should never need to input or output wordLang programs directly (e.g. Graph Analysis should only see a clash graph and the list of moves to be coalesced). The correctness theorem for the entire pass therefore critically rests on the correctness theorem established from Liveness Analysis.

To decouple Liveness Analysis from the other steps, we shall first show that the we can define a property of **suitable** colouring functions that preserves semantics of input programs when applied. Suitability is defined based on the liveness equations: intuitively, the colouring function applied to any clash set of the program must give distinct outputs for that set.

From the other direction, the Graph Colouring step shall be proven to generate a **satisfactory** colouring of the clash graph i.e. no two variables sharing an edge are given the same colour.

We then combine these two theorems to get our final correctness theorem by proving a **linking theorem** between suitable colouring functions and satisfactory colouring functions. Note that Graph Analysis is conspicuously absent from these proofs.

3.1.3 Decoupling Graph Analysis

For the second decoupling, we shall take the view that Graph Analysis is a **heuristic** for Graph Colouring. The key insight here is that we can force Graph Colouring (i.e. Select) to be correct for any input stack ordering (including the one generated by Graph Analysis). This is possible because for each vertex in the stack, we either choose to spill or to colour it with a free colour; a poorly chosen stack ordering merely corresponds to poorer but nevertheless correct performance of the resulting code.

This decoupling yields a lot of freedom in performing the Graph Analysis step since we no longer need to guarantee much about the properties of the stack ordering which

it generates. This is exploited to write significantly more complicated Graph Analysis algorithms (e.g. IRC) without compromising the correctness of the entire pass.

3.1.4 Decoupling Colouring Conventions

The final decoupling involves guaranteeing that the final coloured program satisfies `post_alloc_conventions`. Graph Colouring is the only step involved in generating the colouring function. Therefore, we shall prove that the step generates a **conventional** colouring function. We also separately show that applying conventional colouring functions to a program satisfying `pre_alloc_conventions` program turns it into one satisfying `post_alloc_conventions`.

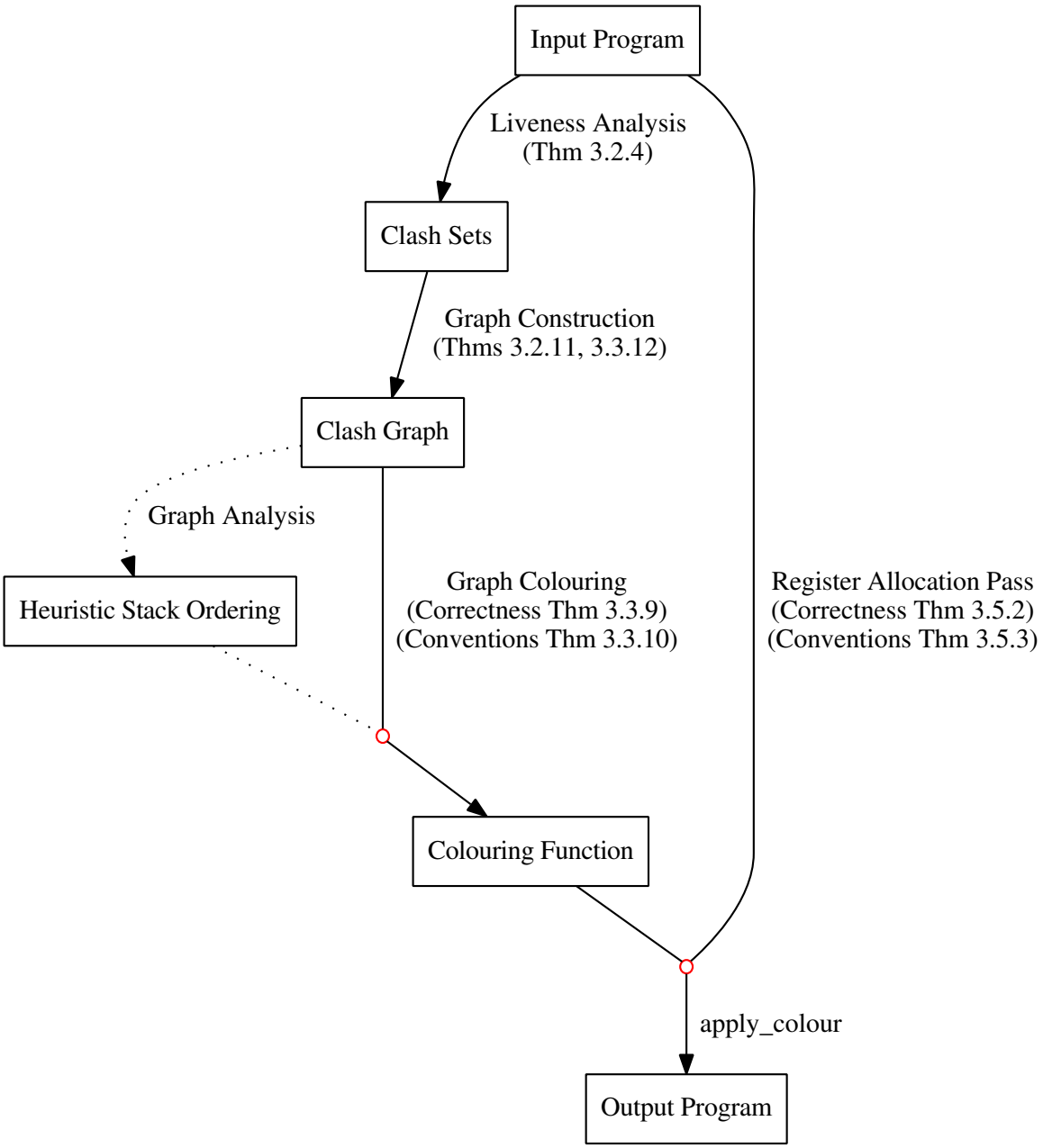

Figure 3.1: Overview of the implementation and proof structure.

3.2 Liveness Analysis

Liveness Analysis amounts to solving Equations 2.3 and 2.4. Since wordLang programs contain no explicit looping commands, their control flow graphs will always correspond directly to their ASTs. This implies that the equations have a direct solution through a bottom up walk of the program's AST[1].

3.2.1 Defining Liveness Analysis

Definition 3.2.1 implements the liveness equations directly. For each *liveout* set (*live*) and program (*prog*), `get_live` *prog live* returns the *livein* set to that program.

Definition 3.2.1. `get_live` (abridged)

```
get_live (Move pri ls) live =
(let killed = FOLDR delete live (MAP FST ls)
 in
   numset_list_insert (MAP SND ls) killed)
get_live (Seq s1 s2) live = get_live s1 (get_live s2 live)
get_live (If cmp r1 ri e2 e3) live =
(let e2_live = get_live e2 live in
 let e3_live = get_live e3 live in
 let union_live = union e2_live e3_live
 in
   case ri of
     Reg r2 ⇒ insert r2 () (insert r1 () union_live)
   | Imm v3 ⇒ insert r1 () union_live)
```

The notion that no two variables in the same clash set has the same colouring can be formalised using the notion of function injectivity over a set.

Definition 3.2.2. Injectivity

$$\texttt{INJ}\ f\ s \iff \forall x\ y.\ x \in s \wedge y \in s \Rightarrow f\ x = f\ y \Rightarrow x = y$$

The key correctness condition (i.e. suitability) is defined for colouring functions via recursion over the structure of the program. Given a colouring function f and program *prog*, `colouring_ok` recursively checks all the clash sets of the program to ensure that f is injective over all of them. Intuitively, `colouring_ok` represents the condition that no two clashing variables get mapped to the same register.

Definition 3.2.3. `colouring_ok` (abridged)[2]

```
colouring_ok f (Move v5 v6) live ⟺
(let lset = get_live (Move v5 v6) live in
 let iset = union (get_writes (Move v5 v6)) live
 in
```

[1] Standard solutions require a fixed point iteration to deal with back edges.

[2] `get_writes` extracts the writes done in that command.

```
    INJ f (domain lset) ∧ INJ f (domain iset))
colouring_ok f (Seq s1 s2) live ⟺
(let s2_live = get_live s2 live in
 let s1_live = get_live s1 s2_live
 in
    INJ f (domain s1_live) ∧ colouring_ok f s2 live ∧
    colouring_ok f s1 s2_live)
colouring_ok f (If cmp r1 ri e2 e3) live ⟺
(let e2_live = get_live e2 live in
 let e3_live = get_live e3 live in
 let union_live = union e2_live e3_live in
 let merged =
       case ri of
         Reg r2 ⇒ insert r2 () (insert r1 () union_live)
       | Imm v3 ⇒ insert r1 () union_live
 in
    INJ f (domain merged) ∧ colouring_ok f e2 live ∧
    colouring_ok f e3 live)
```

3.2.2 Correctness Theorem

We are now ready to prove the correctness theorem: Theorem 3.2.4 states that the evaluation of a coloured program on a **coloured state**, *cst*, gives the same result as evaluating the original program on an **original state**, *st*. The original and coloured states should be thought of as intermediate states during evaluation of their respective programs starting from the same initial state; this generalisation makes the theorem amenable to proof by induction. Note that the quantification over permutation oracles is related to how wordLang fits into the backend and is further discussed in Chapter 4.

Theorem 3.2.4. Liveness Analysis Correctness Theorem

```
⊢ ∀prog st cst f live.
    colouring_ok f prog live ∧ word_state_eq_rel st cst ∧
    strong_locals_rel f (domain (get_live prog live)) st.locals
      cst.locals ⇒
    ∃perm'.
      (let (res,rst) = wEval (prog,st with permute := perm')
       in
         if res = SOME Error then T
         else
           (let (res',rcst) = wEval (apply_colour f prog,cst)
            in
              res = res' ∧ word_state_eq_rel rst rcst ∧
              case res of
                NONE ⇒
                  strong_locals_rel f (domain live) rst.locals
                    rcst.locals
              | SOME v1 ⇒ T))
```

Proof. By structural induction on the form of input programs and expanding the evaluation semantics (`wEval`). This is arguably the most important theorem in this project so the rest of this subsection explains the important cases and how the generalisation works. A brief walkthrough of this proof is presented in Appendix B. □

Simple imperative commands

Most commands of wordLang have fairly simple behaviour: read variables, compute on their values then write-back the result. Since these do not interact with the permutation oracle, we can instantiate the existentially quantified $perm'$ to any permutation desired. For these commands, the proof of correctness is primarily done by direct expansion of the corresponding evaluation semantics.

As a running example, let us consider the code fragment in Table 3.1 (I have left the colouring function unevaluated for illustrative purposes).

Original Program	Coloured Program
$x := 3$	$f(x) := 3$
$y := 2$	$f(y) := 2$
$z := x + y$	$f(z) := f(x) + f(y)$

Table 3.1: Original and Coloured Program Evaluation.

After evaluating the first two lines of both programs, we would expect the corresponding (unordered) locals to look like Figure 3.2.

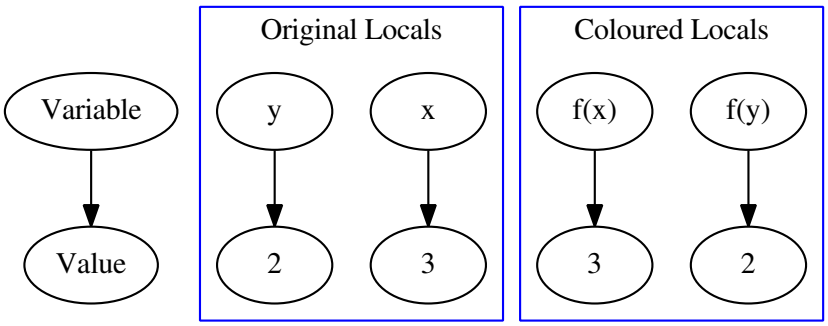

Figure 3.2: Illustration of `strong_locals_rel`.

When we are executing the third line, the original and coloured states clearly have very different local variable names (x and y in the former, $f(x)$ and $f(y)$ in the latter). This motivates the proof generalisation mentioned above; we have to use a **state relation** between the original state (st) and coloured state (cst).

Definition 3.2.5 states that for every live original variable (n) in the original locals ($slocs$), the corresponding coloured variable ($f\ \ n$) in the coloured locals ($tlocs$) should exist and have the same value. This is exactly the case in our example, i.e. `strong_locals_rel` holds between the two intermediate states.

Definition 3.2.5. `strong_locals_rel`

```
strong_locals_rel f ls slocs tlocs ⟺
∀n v. n ∈ ls ∧ lookup n slocs = SOME v ⇒ lookup (f n) tlocs = SOME v
```

Given this assumption, we can see that the third commands become $f(z) := f(x) + f(y) = 5$ and $z := x + y = 5$ respectively. After executing this command, looking up z and $f(z)$ gives the same values in their respective locals. Therefore, `strong_locals_rel` is preserved across execution of these commands. In general, we show that `strong_locals_rel` is preserved across all executions where the result is `NONE`. The proof that this preservation holds requires reasoning about clash sets by unfolding the definitions of `colouring_ok`, `INJ` and using the ideas explained in §2.3.

Since we are only renaming the local variables, the coloured state should only differ from the original state in the locals, stack and permutation oracle state components. We also need a **frame axiom** for the other state components, as defined in Definition 3.2.6.

Definition 3.2.6. `word_state_eq_rel`

```
word_state_eq_rel s t ⟺
t.store = s.store ∧ t.stack = s.stack ∧ t.memory = s.memory ∧
t.mdomain = s.mdomain ∧ t.gc_fun = s.gc_fun ∧ t.handler = s.handler ∧
t.clock = s.clock ∧ t.code = s.code ∧ t.output = s.output
```

`strong_locals_rel` and `word_state_eq_rel` together form the required state relation between original and coloured states. In the theorem statement, we assumed that this state relation holds initially and show that it continues to hold in the conclusion after executing a command. This is usual for an inductive proof because we need to chain the conclusion of one instantiation of the inductive hypothesis to the assumption of another.

Commands requiring induction

There are several wordLang commands that recursively call `wEval` on subprograms in their semantics (e.g. `Seq`, `If`). The main purpose of doing an inductive proof is to use the inductive hypothesis on these subprograms.

We can almost directly prove the inductive cases by instantiating the induction hypothesis appropriately. Unfortunately, the order of quantifications over the permutation oracle is slightly tricky and we also need Theorem 3.2.7 for these cases.

Theorem 3.2.7. Permutation Oracle Swapping Theorem

```
⊢ ∀prog st perm.
    (let (res,rst) = wEval (prog,st)
     in
       res ≠ SOME Error ⇒
       ∃perm'.
         wEval (prog,st with permute := perm') =
         (res,rst with permute := perm))
```

Proof. By induction on the evaluation semantics. Intuitively, this is true because in any terminating evaluation of a program we will only use up a finite part of the permutation oracle. The program behaves identically even if we swapped the tail of this oracle for any other tail[3]. □

Commands interacting with the stack

Finally, we need to provide appropriate oracle permutations whenever a command constructs stack frames for the original/coloured variables. The order of quantification used means that we need to find an oracle for the original state that permutes the original stack frame to match the coloured stack frame.

Lemma 3.2.8 shows that we can always find such an oracle; its conclusion, `MAP` $(\lambda\,(x,y).\ (f\ x,y))\ l' = l$, means that remapping all the variable names of the original stack frame l' with f gives us the coloured stack frame l.

Lemma 3.2.8. Existence of a suitable oracle permutation

```
domain y = IMAGE f (domain x) ∧ INJ f (domain x) ∧
strong_locals_rel f (domain x) x y ⇒
(let (l,permute) = env_to_list y perm
 in
   ∃ perm'.
     (let (l',permute') = env_to_list x perm'
      in
        permute' = tperm ∧ MAP (λ(x,y). (f x,y)) l' = l))
```

Proof. The stack frame construction function, `env_to_list`, is essentially a series of bijective functions. Figure 3.3 shows how this works; the top and bottom paths correspond to `env_to_list` on the original and coloured locals respectively[4]. The bijection we need to produce is shown with a dotted edge; it can be constructed by taking appropriate compositions of the other functions and function inverses since they are all bijections. □

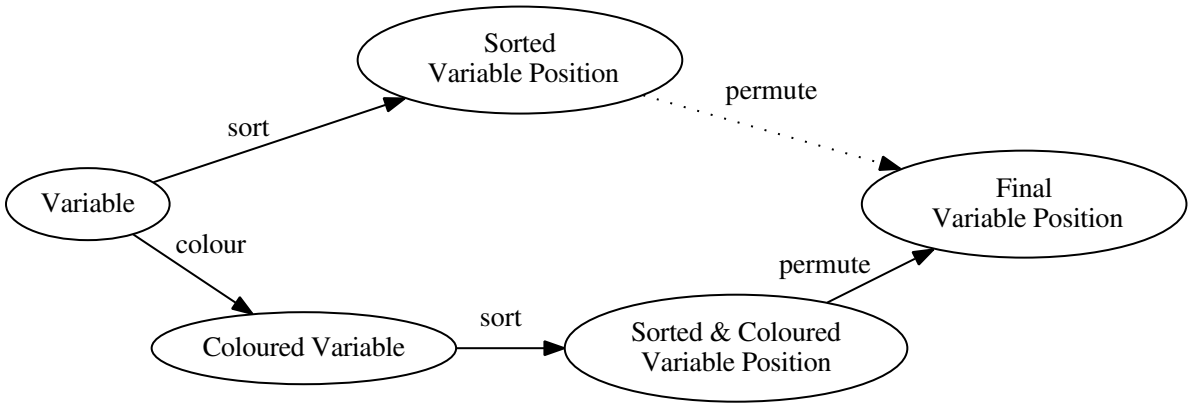

Figure 3.3: Series of bijections between variables and variable stack positions.

[3]Recall that permutation oracles are infinite sequences of bijections.

[4]Note that the colouring function is bijective in this case because of a special property of the semantics.

3.2.3 Extraction of Clash Sets

The `colouring_ok` condition defines an **abstract** property over programs. Its recursive definition over the structure of input programs makes Theorem 3.2.4 provable by direct structural induction. For the next step of the register allocator, we need a **concrete** list of clash sets to work with.

We first notice that the clash graph does not require any control flow information associated with clash sets; it is sufficient to extract all the clash sets of our program into an unordered list. Definition 3.2.9 is quite similar to Definition 3.2.3 except we now concretely accumulate a list of clash sets.

Definition 3.2.9. `get_clash_sets` (abridged)

```
get_clash_sets (Move v3 v4) live =
(let i_set = union (get_writes (Move v3 v4)) live
 in
   (get_live (Move v3 v4) live,[i_set]))
get_clash_sets (Seq s1 s2) live =
(let (hd,ls) = get_clash_sets s2 live in
 let (hd',ls') = get_clash_sets s1 hd
 in
   (hd',ls' ++ ls))
get_clash_sets (If cmp r1 ri e2 e3) live =
(let (h2,ls2) = get_clash_sets e2 live in
 let (h3,ls3) = get_clash_sets e3 live in
 let union_live = union h2 h3 in
 let merged =
       case ri of
         Reg r2 ⇒ insert r2 () (insert r1 () union_live)
       | Imm v3 ⇒ insert r1 () union_live
 in
   (merged,ls2 ++ ls3))
```

Correspondingly, Definition 3.2.10 is an alternative concrete correctness condition that checks the injectivity property over all clash sets in the list generated by `get_clash_sets`.

Definition 3.2.10. `colouring_ok_alt`

```
colouring_ok_alt f prog live ⟺
(let (hd,ls) = get_clash_sets prog live
 in
   EVERY (λs. INJ f (domain s)) ls ∧ INJ f (domain hd))
```

Theorem 3.2.11 links us back to the original theorem by showing that satisfying the concrete `colouring_ok_alt` implies satisfying the abstract `colouring_ok`.

Theorem 3.2.11. Correctness of Concretisation

```
⊢ ∀f prog live.
    colouring_ok_alt f prog live ⇒ colouring_ok f prog live
```

Proof. By case analysis on the induction theorem for `colouring_ok` and observing that every clash set of the program must appear somewhere (not necessarily in program order) in the list of clash sets. □

The following lemma shows a useful property of clash sets (`get_clash_sets`) which is not true of live sets.

Lemma 3.2.12. Every variable appears in some clash set

```
⊢ ∀prog live.
    (let (hd,clash_sets) = get_clash_sets prog live in
     let ls = hd::clash_sets
     in
       (∀x. x ∈ domain live ⇒ in_clash_sets ls x) ∧
       every_var (in_clash_sets ls) prog)
```

Proof. By structural induction on the form of input programs, noting that every variable that appears is either read or written (or both). Read variables always appear in the live set before an instruction while written variables always appear in the clash set after an instruction. □

3.3 Graph Colouring

This section looks at how Graph Colouring is defined so that we achieve the desired decoupling of Graph Analysis from it. A useful technique throughout this section is the tradeoff between implementation complexity and proof complexity. It is often easier to prove that a property holds if we perform **explicit checks** on that property in the implementation instead of relying on invariants. This usually adds a small constant factor to the implementation but significantly reduces proof effort.

3.3.1 Construction of Clash Graphs

The function `clash_sets_to_sp_g` takes a list of clash sets and inserts edges between vertices in the same clash set into the clash graph. This is a direct construction from the definition of a clash graph.

Lemmas 3.3.1 and 3.3.2 are useful properties of this construction that are used later to link up the proofs.

Lemma 3.3.1. Clash sets appear as clash graph cliques

```
⊢ ∀ls x.
    MEM x ls ⇒
    sp_g_is_clique (MAP FST (toAList x)) (clash_sets_to_sp_g ls)
```

Proof. By definition of the clash graph. □

Lemma 3.3.2. Clash set variables are vertices of the clash graph

```
⊢ ∀ls x. in_clash_sets ls x ⇒ x ∈ domain (clash_sets_to_sp_g ls)
```

Proof. By definition of the clash graph. □

3.3.2 Graph Colouring Properties

The Graph Colouring step must produce a colouring function with two key properties (Definitions 3.3.3 and 3.3.4). The first property states that no two adjacent vertices of the clash graph have the same colour (i.e. it is satisfactory) while the latter guarantees that the colouring function is conventional.

Definition 3.3.3. `colouring_satisfactory`

```
colouring_satisfactory col G ⟺
∀v.
  v ∈ domain G ⇒
  (let edges' = lookup v G
   in
     case edges' of
       NONE ⇒ F
     | SOME edges ⇒ ∀e. e ∈ domain edges ⇒ col e ≠ col v)
```

Definition 3.3.4. `colouring_conventional`

```
colouring_conventional G k col ⟺
(let vertices = domain G
 in
   ∀x.
     x ∈ vertices ⇒
     if is_phy_var x then col x = x
     else
       (let y = col x
        in
          if is_stack_var x then is_phy_var y ∧ y ≥ 2 × k
          else is_phy_var y))
```

3.3.3 Defining the Colouring Algorithm

The Graph Colouring step takes an input stack and for each variable in the stack, either assigns it a colour or spills the variable. This is defined over an input list (representing a stack) in Definition 3.3.5. For each member of the list, we either extend the finite partial function *col* with a new colouring or push the vertex onto the spills list *spills*.

Definition 3.3.5. `alloc_colouring_aux`

```
alloc_colouring_aux G k prefs [] col spills = (col,spills)
alloc_colouring_aux G k prefs (x::xs) col spills =
(let (col,spills) = assign_colour G k prefs x col spills
 in
   alloc_colouring_aux G k prefs xs col spills)
```

Since we want to prove colouring correctness over all possible input lists, we need to ensure that the colouring produced is correct even if the input list is **malformed**, i.e. :

1. Lists containing repeated vertices.

2. Lists containing vertices not in the graph.

3. Lists that do not contain all the vertices of the graph.

The first two malformations can be solved as follows: whenever we call `assign_colour` on x, we check that it has not already been coloured or spilled and that it is actually in the clash graph. If any of these checks fail, we simply ignore that vertex.

The last malformation can be solved by explicitly passing in all the vertices of the graph in an arbitrary order to `alloc_colouring_aux` a second time. Any vertex in the graph that was already in the original list will be skipped because of the repetition check but any that was not present will be coloured.

We also pass a **preference oracle**, *prefs*, as input to `assign_colour`. If we can colour a vertex with one or more colours, the preference oracle is called to select a preferred colour. For example, a preference oracle that does biased selection would check the list of available moves to select a colour that coalesces moves for the vertex being coloured. These ideas are all shown in Definition 3.3.6; note that the accumulator arguments, *col* and *spills*, are left unchanged if a malformation is detected.

Definition 3.3.6. `assign_colour`

```
assign_colour G k prefs x col spills =
case lookup x col of
  NONE ⇒
    if MEM x spills then (col,spills)
    else
      (case lookup x G of
         NONE ⇒ (col,spills)
       | SOME colour ⇒
           if is_alloc_var x then
             (let xs = MAP FST (toAList colour) in
              let k = remove_colours col xs k
              in
                case k of
                  [] ⇒ (col,x::spills)
                | colour::xs ⇒
                    (let colour =
                           case prefs x (colour::xs) col of
                             NONE ⇒ colour
                           | SOME y ⇒ y
                     in
                       (insert x colour col,spills)))
           else (col,x::spills))
| SOME x ⇒ (col,spills)
```

3.3.4 Correctness and Conventions Theorems

To show that we are generating `colouring_satisfactory` colourings, we first generalise to the case where we are midway through colouring the graph. Definition 3.3.7 shows the generalised form, where *col* is a partially constructed colouring and we only check the satisfactory condition on the subgraph induced by its domain of definition. Lemma 3.3.8 is the corresponding generalised correctness theorem.

Definition 3.3.7. `partial_colouring_satisfactory`

```
partial_colouring_satisfactory col G ⟺
domain col ⊆ domain G ∧
∀ v .
  v ∈ domain G ∧ v ∈ domain col ⇒
  (let edges = THE (lookup v G)
   in
     ∀ v' .
       v' ∈ domain edges ∧ v' ∈ domain col ⇒
       lookup v col ≠ lookup v' col)
```

Lemma 3.3.8. `alloc_colouring_aux` produces partially satisfactory colourings

```
⊢ ∀ G k prefs ls col spill .
    undir_graph G ∧ satisfactory_pref prefs ∧
    partial_colouring_satisfactory col G ⇒
    (let (col',spill') = alloc_colouring_aux G k prefs ls col spill
     in
       partial_colouring_satisfactory col' G)
```

Proof. By list induction, using `partial_colouring_satisfactory` as the inductive invariant. □

Theorem 3.3.9 shows the final colouring correctness theorem. Note that `irc_alloc` defines the entire allocator, including the Graph Analysis, Graph Colouring and the Spill Allocation phase.

Theorem 3.3.9. IRC Correctness Theorem

```
⊢ ∀ G k moves .
    undir_graph G ⇒
    (let col = irc_alloc G k moves
     in
       colouring_satisfactory (total_colour col) G)
```

Proof. Specialising Lemma 3.3.8 down to the case where we have finished colouring the entire graph. The proof is independent (with a small caveat) of the properties of Graph Analysis. □

Theorem 3.3.10 shows the final conventions theorem. It has the same form as Theorem 3.3.9 and a similar approach is used to prove it.

Theorem 3.3.10. IRC Conventions Theorem

```
⊢ ∀ G k moves.
    undir_graph G ⇒
    (let col = irc_alloc G k moves
     in
       colouring_conventional G k (total_colour col))
```

Proof. By defining a suitable conventions predicate for partial colourings, proving the corresponding generalised theorem and specialising it down to the full colouring. □

3.3.5 Linking to Liveness Analysis

To link us back to the Liveness Analysis, we now need to show that any colouring that satisfies `colouring_satisfactory` on a clash graph also satisfies `colouring_ok_alt` on the clash set list.

Recall that Lemma 3.3.1 allows us to reduce reasoning about clash sets to reasoning about cliques. This motivates Lemma 3.3.11.

Lemma 3.3.11. Clique vertices are always assigned distinct colours

```
⊢ ∀ ls g f.
    ALL_DISTINCT ls ∧ colouring_satisfactory f g ∧
    sp_g_is_clique ls g ⇒
    ALL_DISTINCT (MAP f ls)
```

Proof. By unfolding the definition of `colouring_satisfactory` on a clique. □

We can now establish the crucial link from Graph Colouring to Liveness Analysis.

Theorem 3.3.12. `colouring_satisfactory` implies `colouring_ok_alt`

```
⊢ ∀ prog f live.
    (let spg = get_spg prog live
     in
       colouring_satisfactory f spg ⇒ colouring_ok_alt f prog live)
```

Proof. Every clash set is a clique (Lemma 3.3.1) and every clique vertex is assigned distinct colours (Lemma 3.3.11). □

3.4 Graph Analysis

My implementation of the IRC algorithm largely follows the **worklists** implementation by Appel [1]. The central idea is to use some book-keeping to track the vertices that can be Simplified, Coalesced, Frozen or Spilled in future iterations. This is more efficient than searching through the entire graph each time.

Unfortunately, his implementation is imperative and has to be adapted for the purely functional HOL4 definition language. For example, $O(1)$ mutable arrays are not available so I have replaced them with $O(\log n)$ functional arrays. Some changes were also needed to fit the standard algorithm into the proof framework.

3.4.1 Termination

The IRC algorithm repeatedly extracts vertices from the graph until it is empty. It is not immediately obvious that the algorithm terminates, especially in the worklists implementation.

A way to simplify the termination proof is to take the view that every iteration must pick exactly one vertex from the graph and put it onto the colouring stack. Each of the possible steps of the algorithm can be tweaked to work in this way:

- Simplify/Spill are already in this form.
- Freeze can be rephrased to immediately Simplify the frozen vertex since this will be done in a future pass anyway.
- Instead of explicitly merging two coalesceable vertices u and v, Coalesce will instead place vertex v on the stack and add all the edges of v to u. Since u remains in the graph, it will eventually get pushed onto the stack and therefore, it will necessarily appear higher in the colouring stack than v. By the time we reach v, we can always assign it the same colour that u has already been assigned[5]. Figure 3.4 shows an example of u and v being coalesced (the dotted edge) and the corresponding changes to the graph/stack.

With this change, the number of iterations that we need to perform becomes exactly the number of vertices in the input graph i.e. termination is trivial.

3.4.2 Monadic Implementation

The IRC algorithm is defined with a state monad to more closely match its original imperative implementation. Definition 3.4.1 shows the monadic implementation of Simplify; it returns `NONE` if Simplify fails and `SOME` x otherwise (where x is to be put onto the colouring stack). The other steps are implemented similarly.

Definition 3.4.1. `simplify`

```
simplify =
do
  simps ← get_simp_worklist;
  case simps of
    [] ⇒ return NONE
  | x::xs ⇒
      do set_simp_worklist xs; dec_deg x; unspill; return (SOME x) od
od
```

We also add a **clock** component to our monad that tracks how many iterations are left. The clock is decremented in each iteration and we terminate the algorithm when it hits 0. This allows us to use induction on the value of the clock to prove simple invariants

[5]Some extra book-keeping is required to track that u and v should be assigned the same colour.

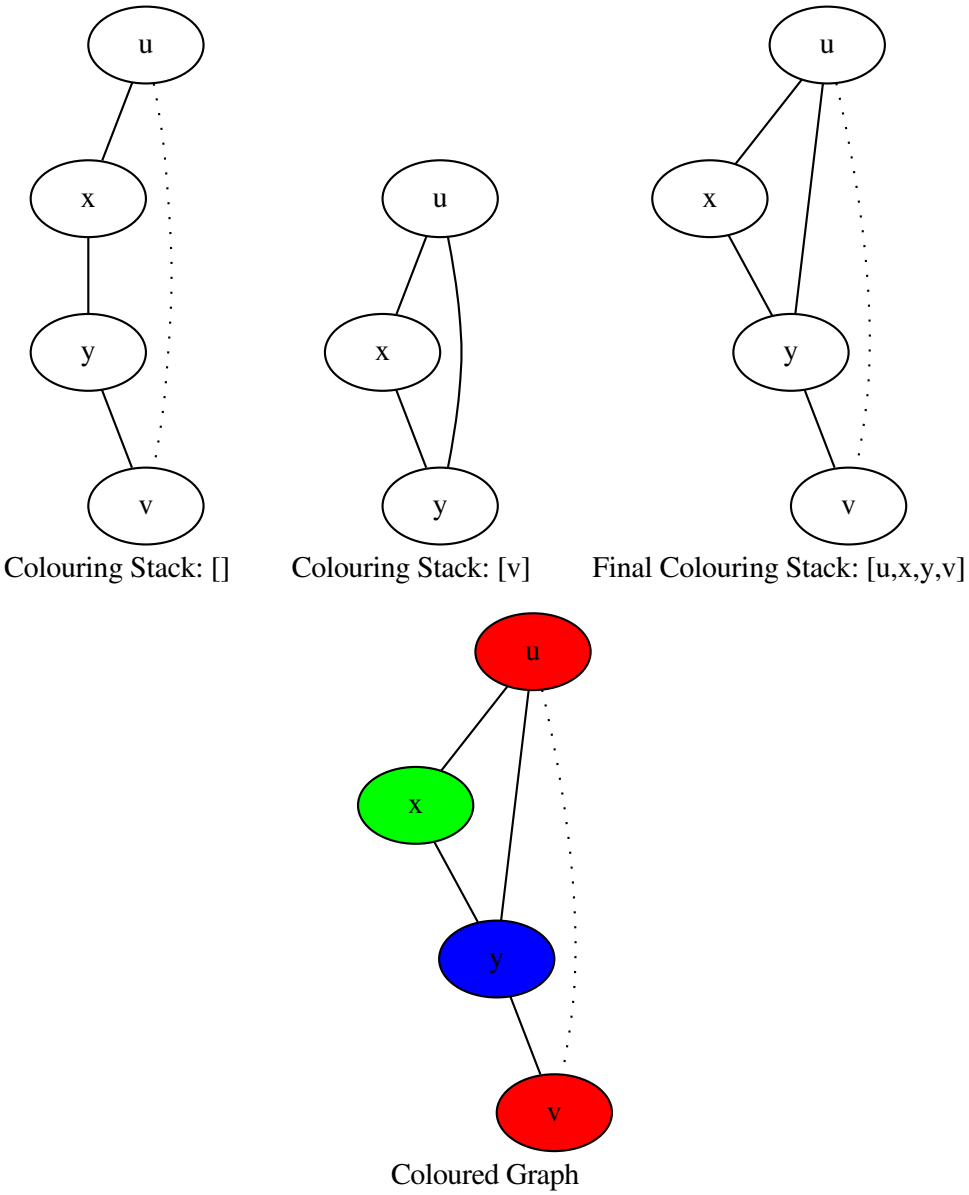

Figure 3.4: Coalescing of vertices in the clash graph.

about Graph Analysis. Definition 3.4.2 shows the definition of a single iteration of the IRC; notice that it only calls `coalesce` if `simplify` fails, etc. and that it decrements the clock at the start.

Definition 3.4.2. `do_step`

```
do_step =
do
  dec_clock;
  optx ← simplify;
  case optx of
    NONE ⇒
      do
        optx ← coalesce;
        case optx of
          NONE ⇒
            do
              optx ← freeze;
              case optx of
                NONE ⇒
                  do
                    optx ← spill;
                    case optx of
                      NONE ⇒ return ()
                    | SOME x ⇒ push_stack x
                  od
              | SOME x ⇒ push_stack x
            od
        | SOME x ⇒ push_stack x
      od
  | SOME x ⇒ push_stack x
od
```

The entire IRC loop is defined in Definition 3.4.3 using usng a monadic while (`MWHILE`). The predicate `has_work` checks the clock condition in each iteration.

Definition 3.4.3. `rpt_do_step`

```
rpt_do_step = MWHILE has_work do_step
```

3.4.3 Handling Coalesced Vertices

The definition of Graph Colouring does not allow us to forcibly assign two vertices the same colour when we do coalescing. We can achieve an approximation to forced assignment by providing a preference oracle that always tries to pick the coalesced colour before trying Biased Selection.

The distinction is a purely semantic trick: if the algorithms were implemented correctly, then all the coalesces are necessarily valid and the coalesced colour will always be a possible choice for the oracle. Indirecting through the preference oracle allows us to skip this (difficult) proof.

3.4.4 Handling Extended Graphs

Coalescing might add edges to the graph that need to be taken into account during Graph Colouring. To prevent this from breaking the proof decoupling, we first notice that the Coalesce step (and therefore the entire Graph Analysis) will only ever add edges to the graph. A weak notion of subgraph in Definition 3.4.4; every edge $x \to y$ in G is also an edge in H.

Definition 3.4.4. `is_subgraph_edges`

```
is_subgraph_edges G H ⟺
domain G = domain H ∧ ∀x y. lookup_g x y G ⇒ lookup_g x y H
```

Lemma 3.4.5 shows that `is_subgraph_edges` is a useful invariant for Graph Analysis (the input and output graphs are always related by it).

Lemma 3.4.5. Graph Analysis always extends input graph; i.e. the initial graph, s.`graph`, is a subgraph of the final graph, s'.`graph`

```
⊢ ∀s.
    undir_graph s.graph ⇒
    (let ((),s') = rpt_do_step s
     in
       undir_graph s'.graph ∧ is_subgraph_edges s.graph s'.graph)
```

Proof. By induction on the clock and expanding the monadic definitions. □

Correspondingly, Lemma 3.4.6 shows that the colouring satisfactory property holds for all subgraphs. This directly implies that if we apply Graph Colouring on a clash graph produced by Graph Analysis, then the resulting colouring is also satisfactory for the original clash graph.

Lemma 3.4.6. `partial_colouring_satisfactory` applies for all subgraphs

```
⊢ ∀G H col.
    undir_graph G ∧ is_subgraph_edges G H ∧
    partial_colouring_satisfactory col H ⇒
    partial_colouring_satisfactory col G
```

Proof. By observing that subgraph vertices are always less constrained so any colouring of H works for the subgraph, G. □

These lemmas, along with our proof decoupling, allow Graph Analysis to be integrated into the proofs of Theorem 3.3.9 and Theorem 3.3.10 easily.

3.5 Register Allocation Pass

We now have all the components necessary to define and verify the entire register allocation pass, as shown in Definition 3.5.1.

Definition 3.5.1. `word_alloc`

```
word_alloc k prog =
(let clash_graph = get_spg prog LN in
 let moves = get_prefs prog [] in
 let col = irc_alloc clash_graph k moves
 in
   apply_colour (total_colour col) prog)
```

The final theorems produced for this pass are the correctness theorem, Theorem 3.5.2 and the conventions theorem, Theorem 3.5.3.

Theorem 3.5.2. Register Allocation Correctness Theorem

```
⊢ ∀prog k st.
    even_starting_locals st.locals ⇒
    ∃perm'.
      (let (res,rst) = wEval (prog,st with permute := perm')
       in
         if res = SOME Error then T
         else
           (let (res',rcst) = wEval (word_alloc k prog,st)
            in
              res = res' ∧ word_state_eq_rel rst rcst))
```

Proof. The conclusion of Theorem 3.3.9 shows that the colouring function produced by `irc_alloc` satisfies `colouring_satisfactory`. We then follow the chain of implications from `colouring_satisfactory` to `colouring_ok` via Theorems 3.3.12 and 3.2.11. Finally, Theorem 3.2.4 is specialised with this colouring function. □

Theorem 3.5.3. Register Allocation Conventions Theorem

```
⊢ ∀prog k.
    pre_alloc_conventions prog ⇒
    post_alloc_conventions k (word_alloc k prog)
```

Proof. Every variable in the program is a vertex of the clash graph by Lemmas 3.2.12 and 3.3.2. Theorem 3.3.10 is then applied with some case analysis to show that the required conventions hold. □

3.6 SSA/CC Pass

Theorem 3.5.3 depends on input code already satisfying `pre_alloc_conventions`. One way to establish these conventions is to have an additional pass before register allocation

that sets up the code in this way. This is advantageous because other passes coming before it do not have to know about these conventions at all (i.e. they are fully decoupled from the allocator).

We also integrate a **static single assignment (SSA)** transformation into the pass. Code in SSA form has at most one write to any variable; subsequent writes to the same variable are renamed uniformly throughout the code. Putting code in SSA form splits up the live range of a variable so each variable generally clashes with fewer other variables; this places less constraints on the register allocator.

The combined pass is called the **SSA/CC pass** and it is a variable renaming transformation within wordLang.

3.6.1 Core Algorithm for SSA/CC

To implement the SSA form algorithm, we keep a **SSA mapping** of variables to their latest assigned names. The mapping is updated as we do a forward walk of the AST, renaming variables appropriately. This motivates Definition 3.6.1 which takes an input program, the current SSA mapping (ssa) and a fresh variable name (na) and returns a tuple consisting of the program in SSA form, the new SSA mapping and the next fresh name to use.

To setup the conventions, `ssa_cc_trans` will also explicitly move variables into appropriate locations. In the fragment of the definition given, it sets up the convention that the argument to Raise will always be variable 2.

Definition 3.6.1. `ssa_cc_trans` (`Move`, `Seq` and `Raise`)

```
ssa_cc_trans (Move pri ls) ssa na =
(let ls_1 = MAP FST ls in
 let ls_2 = MAP SND ls in
 let ren_ls_2 = MAP (option_lookup ssa) ls_2 in
 let (ren_ls_1,ssa',na') = list_next_var_rename ls_1 ssa na
 in
   (Move pri (ZIP (ren_ls_1,ren_ls_2)),ssa',na'))
ssa_cc_trans (Seq s_1 s_2) ssa na =
(let (s_1',ssa',na') = ssa_cc_trans s_1 ssa na in
 let (s_2',ssa'',na'') = ssa_cc_trans s_2 ssa' na'
 in
   (Seq s_1' s_2',ssa'',na''))
ssa_cc_trans (Raise num) ssa na =
(let num' = option_lookup ssa num in
 let mov = Move 0 [(2,num')]
 in
   (Seq mov (Raise 2),ssa,na))
```

Table 3.2 demonstrates the key features of `ssa_cc_trans` (the mapping is shown in [brackets]). Notice that the allocator has given the two original uses of x different register assignments and that it has coalesced the move ($t_2 := x$) introduced to set up the conventions.

Input Code	Calling Conventions	SSA Form	Register Allocation
$z := 5$ $x := 9$ $y := x * z$ $x := y + 10$ `raise` x	$z := 5$ $x := 9$ $y := x * 2$ $x := y + 10$ $t_2 := x$ `raise` t_2	$[\,]$ $z_0 := 5$ $[z \to z_0]$ $x_0 := 9$ $[x \to x_0]$ $y_0 := x_0 * 2$ $[x \to x_0, y \to y_0]$ $x_1 := y_0 + 10$ $[x \to x_1, y \to y_0]$ $t_2 := x_1$ `raise` t_2	$r_0 := 5$ $r_1 := 9$ $r_1 := r_1 * r_0$ $r_2 := r_1 + 10$ $r_2 := r_2$ `raise` r_2

Table 3.2: Transforming input code using the SSA/CC transformation.

3.6.2 Simplification for Branching Control Flow

The SSA form shown so far does not work with branching control flow; a variable might be renamed in different ways on separate branches. The standard solution to handle branching is to introduce ϕ**-functions** at points where branches meet. These functions can be thought of as oracles that know which branch an assignment came from and picks out the correct instance to use. The semantics of wordLang does not currently have ϕ-functions[6].

One simple solution is to **eliminate** the ϕ-functions by changing them into `Move` commands. Definition 3.6.2 shows how this is defined for If commands; `fix_inconsistencies` generates the required `Move` commands, *e2_cons* and *e3_cons* to force the SSA mappings to become consistent. These are executed at the end of their respective branches before control is joined.

Definition 3.6.2. `ssa_cc_trans` (`If`)

```
ssa_cc_trans (If cmp r1 ri e2 e3) ssa na =
(let r1' = option_lookup ssa r1 in
 let ri' =
       case ri of Reg r ⇒ Reg (option_lookup ssa r) | Imm v ⇒ Imm v
 in
 let (e2',ssa2,na2) = ssa_cc_trans e2 ssa na in
 let (e3',ssa3,na3) = ssa_cc_trans e3 ssa na2 in
 let (e2_cons,e3_cons,na_fin,ssa_fin) =
       fix_inconsistencies ssa2 ssa3 na3
 in
   (If cmp r1' ri' (Seq e2' e2_cons) (Seq e3' e3_cons),ssa_fin,
    na_fin))
```

[6] Although we have given some thought to adding it, the semantics is rather complex to deal with.

This solution unfortunately means that our code is no longer strictly in SSA form. Nevertheless, we will still get some benefit from the sequential portions of the code. Table 3.3 shows an example of this elimination.

Input Code	Standard SSA Form	ϕ Elimination
`if` P `then` $x := 5$ `else` $x := 3$ $z := x$	`if` P `then` $x_0 := 5$ `else` $x_1 := 3$ $x_2 := \phi(x_0, x_1)$ $z_0 := x_2$	[ ] `if` P `then` $x_0 := 5;\ x_2 := x_0$ $[x \to x_2]$ `else` $x_1 := 3;\ x_2 := x_1$ $[x \to x_2]$ $z_0 := x_2$

Table 3.3: Dealing with branches in SSA form.

3.6.3 Correctness Theorem

The correctness theorem needed for this pass, Theorem 3.6.3, is very similar in form to Theorem 3.2.4. The idea here is again to generalise to the case where we are midway through evaluation of a program that has been put into SSA form.

Theorem 3.6.3. SSA/CC correctness

```
⊢ ∀prog st cst ssa na.
    word_state_eq_rel st cst ∧
    ssa_locals_rel na ssa st.locals cst.locals ∧ is_alloc_var na ∧
    every_var (λx. x < na) prog ∧ ssa_map_ok na ssa ⇒
    ∃perm'.
      (let (res,rst) = wEval (prog,st with permute := perm')
       in
         if res = SOME Error then T
         else
           (let (prog',ssa',na') = ssa_cc_trans prog ssa na in
            let (res',rcst) = wEval (prog',cst)
            in
              res = res' ∧ word_state_eq_rel rst rcst ∧
              case res of
                NONE ⇒ ssa_locals_rel na' ssa' rst.locals rcst.locals
              | SOME v1 ⇒ T))
```

Proof. By structural induction on the form of input programs and expanding the definition of `ssa_cc_trans`. The purpose of `ssa_locals_rel` mirrors that of `strong_locals_rel` in Theorem 3.2.4: it relates the locals when we are midway through evaluation of the program in SSA form. Since this is also a variable renaming transformation, `word_state_eq_rel` is sufficient to form the rest of our generalised state relation.

Note that the theorem statement contains the following conditions that do not appear in the conclusion:

$$\texttt{is\_alloc\_var}\ na \wedge \texttt{ssa\_map\_ok}\ na\ ssa \wedge \texttt{every\_var}\ (\lambda x.\ x < na)\ prog$$

Informally, these are **freshness preconditions** used to ensure freshness of the names used by the transformation. In order for an inductive proof to succeed, we need to prove that these hold in the conclusion as well. This preservation lemma is independent of the evaluation semantics and can be proven separately in Lemma 3.6.4. □

Lemma 3.6.4. SSA/CC transformation preserves freshness

$$\begin{array}{l}\vdash \forall\, prog\ ssa\ na\ prog'\ ssa'\ na'.\\ \quad \texttt{ssa\_cc\_trans}\ prog\ ssa\ na = (prog', ssa', na') \wedge \texttt{ssa\_map\_ok}\ na\ ssa \wedge\\ \quad \texttt{is\_alloc\_var}\ na \Rightarrow\\ \quad na \leq na' \wedge \texttt{is\_alloc\_var}\ na' \wedge \texttt{ssa\_map\_ok}\ na'\ ssa'\end{array}$$

Proof. By case analysis on the induction theorem for `ssa_cc_trans` and using arithmetic decision tactics. □

3.6.4 Conventions Theorem

The motivating purpose of this pass was to set up the conventions for the register allocator; this is verified in Theorem 3.6.5.

Theorem 3.6.5. SSA/CC conventions

$$\begin{array}{l}\vdash \forall\, prog\ ssa\ na.\\ \quad \texttt{is\_alloc\_var}\ na \wedge \texttt{ssa\_map\_ok}\ na\ ssa \Rightarrow\\ \quad (\textbf{let}\ (prog', ssa', na') = \texttt{ssa\_cc\_trans}\ prog\ ssa\ na\\ \quad\ \textbf{in}\\ \quad\quad \texttt{pre\_alloc\_conventions}\ prog')\end{array}$$

Proof. By structural induction on input programs and unfolding the definition of `ssa_cc_trans`. Lemma 3.6.4 is useful here but most of the work is spent checking that every condition of `pre_alloc_conventions` gets satisfied by the transformation. □

3.6.5 Theorem Specialisation

Having proven the core theorems about `ssa_cc_trans`, we now define an additional step on top of it to remove the freshness preconditions that were used for generalisation. Definition 3.6.6 shows how this is defined; `setup_ssa` and `limit_var` generate generate the appropriate freshness preconditions for *ssa* and *na* respectively. For example, to find a fresh variable name for a program, it suffices to find the maximum variable name used in that program and use any suitable name greater than it.

Definition 3.6.6. `full_ssa_cc_trans`

```
full_ssa_cc_trans n prog =
(let lim = limit_var prog in
 let (mov,ssa,na) = setup_ssa n lim prog in
 let (prog',ssa',na') = ssa_cc_trans prog ssa na
 in
   Seq mov prog')
```

Theorem 3.6.7 and Theorem 3.6.8 are the corresponding correctness and conventions theorems for `full_ssa_cc_trans`. They are the final theorems produced for this pass.

Theorem 3.6.7. Full SSA/CC Correctness Theorem

```
⊢ ∀prog st n.
    domain st.locals = set (even_list n) ⇒
    ∃perm'.
      (let (res,rst) = wEval (prog,st with permute := perm')
       in
         if res = SOME Error then T
         else
           (let (res',rcst) = wEval (full_ssa_cc_trans n prog,st)
            in
              res = res' ∧ word_state_eq_rel rst rcst))
```

Proof. Specialising Theorem 3.6.3 with properties of `setup_ssa` and `limit_var`. □

Theorem 3.6.8. Full SSA/CC Conventions Theorem

```
⊢ ∀n prog. pre_alloc_conventions (full_ssa_cc_trans n prog)
```

Proof. Specialising Theorem 3.6.5 with properties of `setup_ssa` and `limit_var`. □

Chapter 4

Evaluation

This chapter evaluates the usefulness of this project for the CakeML compiler backend. This is first done qualitatively by discussing the important properties of the theorems proven (§4.1 and §4.2). The next section looks at applying CakeML's proof producing translation mechanism (§4.3). The final sections quantitatively evaluates the register allocator (§4.4) and the project (§4.5).

4.1 Proof Composition

Ensuring that theorems compose properly is a central theme throughout this project. We now discuss the various ways in which proofs are composed in this project and why they imply that the theorems proved are suitable for the new CakeML compiler backend.

4.1.1 Composition across wordLang Passes

There will be several other transformation passes in wordLang, both before and after register allocation. It is important that the form of the correctness theorem used within wordLang can be composed easily between wordLang passes. This can be demonstrated by combining the two passes written in this project; Definition 4.1.1 performs the SSA/CC transformation followed by register allocation on the input program.

Definition 4.1.1. `word_trans`

`word_trans` *n k prog* `=` `word_alloc` *k* `(full_ssa_cc_trans` *n prog*`)`

Firstly, this composition decouples previous compilation steps from the impure conventions of wordLang. Theorem 4.1.2 shows that any input program is transformed into one satisfying `post_alloc_conventions`. This was one of the motivations for writing the SSA/CC pass.

Theorem 4.1.2. Composed SSA/CC and Register Allocation Conventions Theorem

$\vdash \forall$ *prog n k*. `post_alloc_conventions` *k* `(word_trans` *n k prog*`)`

Proof. By direct composition of Theorems 3.5.3 and 3.6.8. □

More importantly, Theorem 4.1.3 is the composed correctness theorem for `word_trans`. It has the same form as our previous correctness theorems, which shows that the correctness theorems for other wordLang passes can be composed in the same way.

Theorem 4.1.3. Composed SSA/CC and Register Allocation Correctness Theorem

```
⊢ ∀prog n k st.
    domain st.locals = set (even_list n) ⇒
    ∃perm'.
      (let (res,rst) = wEval (prog,st with permute := perm')
       in
         if res = SOME Error then T
         else
           (let (res',rcst) = wEval (word_trans n k prog,st)
            in
              res = res' ∧ word_state_eq_rel rst rcst))
```

Proof. For any permutation oracle, *perm*, to be used for evaluating the program after the composed pass, there exists a corresponding oracle, *perm′*, for which the register allocator is correct by Theorem 3.5.2. For *perm′*, there exists yet another corresponding oracle *perm″* for which the SSA/CC pass was correct by Theorem 3.6.7. This is a suitable witness and we are done. □

4.1.2 Composition with Adjacent Correctness Theorems

We saw in §2.2 that the compilation into wordLang (from BVP) will be ∀-quantified while the compilation theorem out of wordLang (to stackLang) will be ∃-quantified over permutation oracles. These compilation theorems are not yet proven[1], but we can explain informally why our correctness theorem is precisely designed to be composed with them. Suppose that we have composed all the passes in wordLang (as in the previous subsection) and obtained a correctness theorem like Theoreom 4.1.3.

Given a compilation to stackLang, instantiating the outgoing compilation theorem gives us some oracle *perm* for which the compilation is correct. By Theorem 4.1.3, this corresponds to some oracle *perm′* for which `word_trans` is correct. Since the incoming compilation theorem is proven for all oracles, *perm′* therefore corresponds to a correct compilation from BVP. Figure 4.1 illustrates this argument; we are travelling backwards along the compilation arrows in the proof.

4.1.3 Composition within Register Allocation

As we saw in Theorem 3.5.2, the correctness proof for register allocation depends on a composed chain of implications leading from Graph Colouring to Liveness Analysis. This decoupled proof structure has two benefits:

1. As the backend is still in development, it is possible that the semantics of wordLang might need to change in minor ways to accommodate new optimisations. The

[1]Note that their proofs are out of scope for my project.

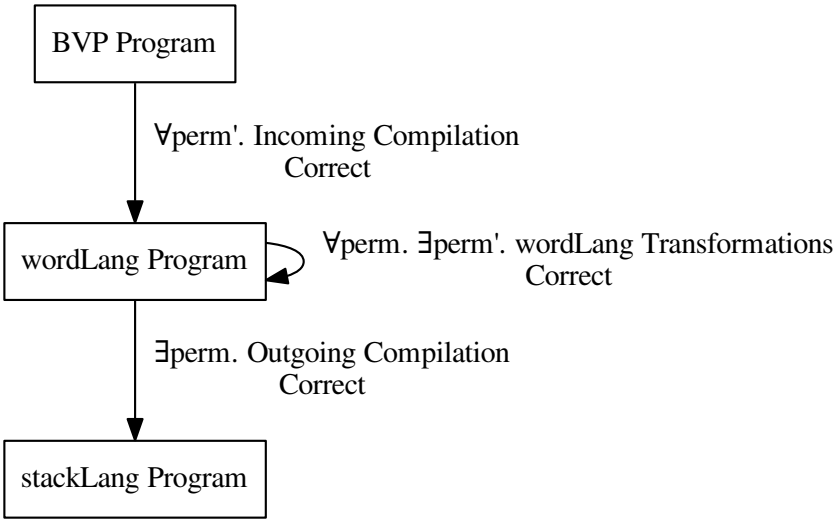

Figure 4.1: Informal composition with adjacent theorems.

Liveness Analysis and corresponding proofs can be changed without affecting Graph Colouring/Graph Analysis at all.

2. Different versions of Graph Analysis can be tried out without affecting the correctness of Graph Colouring (and the rest of the proof).

One example that required changes on both ends of the proof was the modification of the register allocator to support a notion of priorities in coalescing. The ϕ-elimination done in the SSA/CC pass introduces several `Move` commands to the code. The Graph Analysis needed hints that it should prioritise the coalescing of these moves. This was easily implemented without changing the linking theorems at all:

1. The syntax of wordLang was modified to add a priority tag to Move commands.
2. SSA/CC pass was modified to explicitly tag the ϕ-elimination moves.
3. Graph Analysis iteration was modified to preferentially coalesce high priority moves.

4.2 Graph Colouring/Analysis Proof Properties

The Graph Colouring step's correctness proof is completely decoupled from Graph Analysis because we treat the latter as a heuristic. Aside from termination and the subgraph property, there are few theorems about the behaviour of the Graph Analysis step. Unfortunately, this means that the Graph Analysis implementation may contain bugs. However, the only way that these bugs can manifest in output code is slightly poorer performance (e.g. making more spills than necessary) because we have proven that the register allocation pass preserves program semantics. This is acceptable because I aimed to verify the

correctness of the overall register allocation pass and not the correctness of the Graph Analysis algorithm. The following subsections discuss alternative methods of verifying register allocation.

4.2.1 Fully Verified IRC

The IRC algorithm has been formally specified and verified [7]. This shows that it is certainly possible to have a complete verification of the Graph Analysis step. Note, however, that their specification was written in Coq and required significant development of supporting theories, both are out of scope for this project.

Interestingly, the authors encountered a very similar disadvantage with implementing IRC in a theorem prover: the IRC algorithm can be efficiently implemented using mutable state, which is unavailable in a purely functional programming environment.

4.2.2 Translation Validation

An alternative is to use **translation validation** for register allocation [16]. In this approach, the compiler first makes an external call to an unverified register allocator. The result of this unverified register allocation is then passed through a **verified checker** in the compiler[2]. This allows the register allocator to be implemented outside the theorem prover (e.g. it could be written in OCaml, potentially with stateful code) but allows the compiler to remain verified.

A minor deficiency in this approach is if the checker rejects the unverified allocation: in that case, the compilation would not be able to continue even though the input program was fine. My approach side-steps this issue because the Graph Colouring correctness theorem guarantees that we will always produce a proper colouring. Unfortunately, my register allocator is still completely implemented in the theorem prover so it might be a bottleneck in the compilation process. The next section explains how we can switch to translation validation without much difficulty.

4.3 Proof Producing Translation

The CakeML compiler is able to **bootstrap** itself, i.e. produce a verified compiler for CakeML written in CakeML. Since our compiler is written entirely using HOL definitions, we first generate a CakeML AST from these definitions. This AST is then compiled using the original HOL definitions to complete the bootstrap.

The generation step is supported by a powerful **proof producing translation** mechanism [14]. This takes an input HOL definition and produces as output a CakeML AST implementing that definition[3]. Each translation is also accompanied by a certificate theorem for that translation's correctness with respect to the semantics of CakeML.

The entire process is summarised in Figure 4.2; pretty printing is purely cosmetic and is not verified.

[2]The checker could, for example, check that a colouring does not produce clashes in the clash graph.

[3]The input definition usually has to be of a suitable / recognisable form.

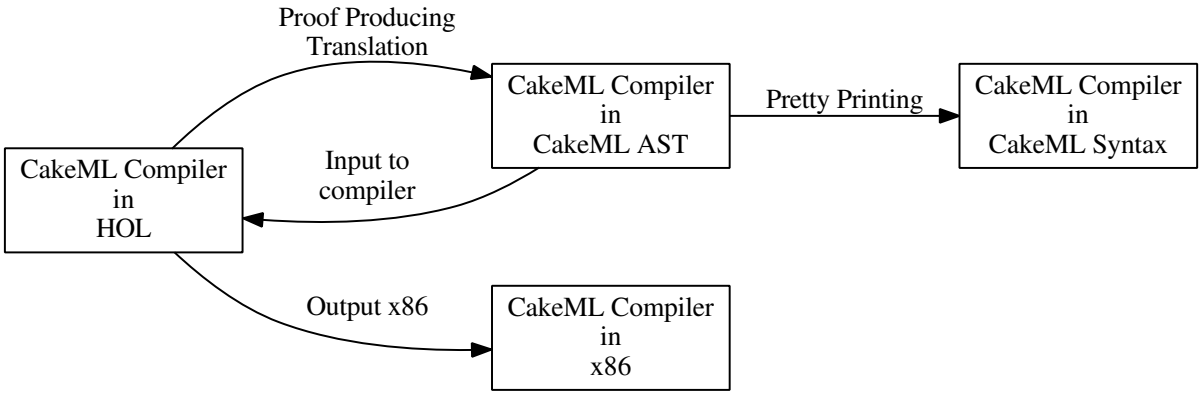

Figure 4.2: Bootstrapping mechanism of the CakeML compiler.

Using this mechanism, I have translated the Graph Analysis and Graph Colouring steps to get a CakeML version of the IRC algorithm. Ensuring that the translation is successful is important because the allocator will eventually be part of the bootstrap process. The main difficulty here was to manually prove some side theorems (e.g. the termination theorem for Graph Analysis) that the proof producing translation mechanism cannot prove automatically.

Pretty printing the resulting CakeML AST yields a version of the register allocator in CakeML syntax (that can also be interpreted as SML). As an example, Definition 4.3.1 is a simple helper function and Figure 4.3 shows its corresponding translation being interpreted in Poly/ML.

Definition 4.3.1. `option_filter`

```
option_filter [] = []
option_filter (x::xs) =
case x of NONE ⇒ option_filter xs | SOME y ⇒ y::option_filter xs
```

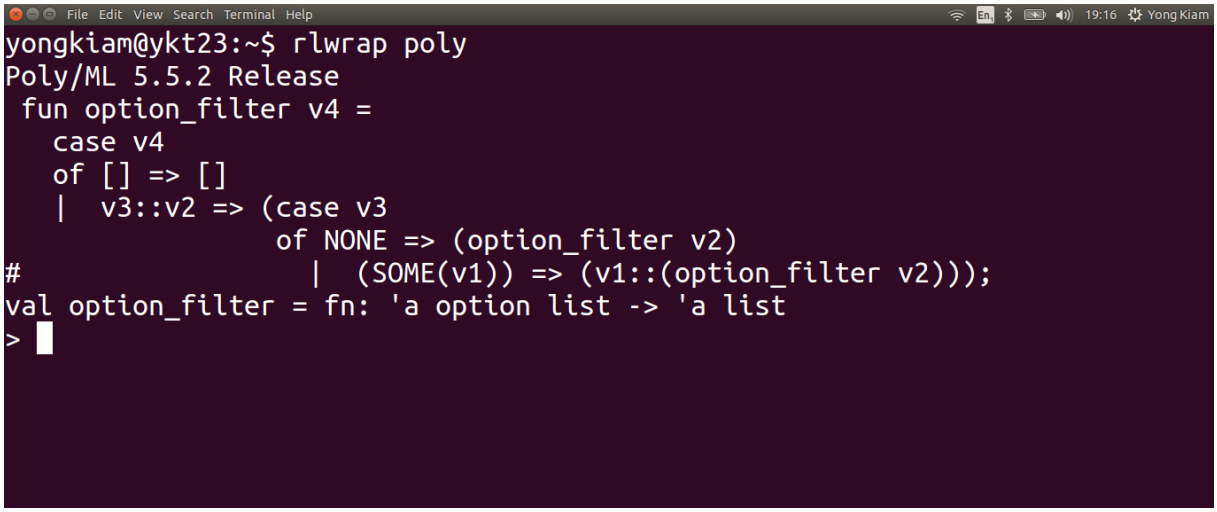

Figure 4.3: Translation of `option_filter`.

The translated allocator can be treated as an unverified SML register allocator (ran in Poly/ML) to be used during compilation. This solves the bottleneck problem mentioned in §4.2.

4.4 Register Allocator Performance

As the new CakeML compiler backend is still under development, it was not possible to do end-to-end performance tests on CakeML programs. I have instead evaluated the register allocator (and its translated version) on a corpus of register allocation problems, focusing on the three optimisation goals mentioned in §1.3.

4.4.1 Evaluation Setup

Appel [2] provides a corpus of 27,922 register allocation problems generated during the bootstrapping of the SML/NJ compiler [5]. Each problem consists of a clash graph and a list of coalesceable moves.

The corpus was first filtered down to 2,357 problems by discarding the ones that have very few coalesceable moves. This is reasonable since the heuristics used will differ only when there is a sizeable number of coalesceable moves. A simple conversion script was used to convert this smaller set of graphs into an input format suitable for Poly/ML and HOL4.

In this reduced set, there are several extremely large graphs (> 500 vertices) that could not be solved in a reasonable amount of time inside HOL4. The problems were further partitioned into 2,222 **small** (≤ 500 vertices) and 135 **large** problems. Evaluation in HOL4 was only performed on the smaller set.

All performance measurements were done inside Poly/ML and HOL4 on the CakeML regression server (64GB RAM, Intel ® Core™ i7-3820 @ 3.60GHz).

4.4.2 HOL4 Evaluation (Small Problems)

The performance of three verified Graph Analysis heuristics were measured and they are labelled as follows[4]:

- **Baseline** - Simplify-Spill with Biased Selection, this should be the fastest algorithm.
- **Briggs** - Briggs-style coalescing, this should provide a tradeoff of coalescing performance for speed.
- **IRC** - The full IRC algorithm.

Figure 4.4 shows plots of the times taken by each heuristic and a superimposed plot of the trend lines for each heuristic. The trend lines are plotted for problems with ≥ 30 vertices since the smaller problems seemed to have some start-up delay.

There are several observations suggested by these plots:

[4]The heuristics are all verified using the techniques developed for Graph Analysis.

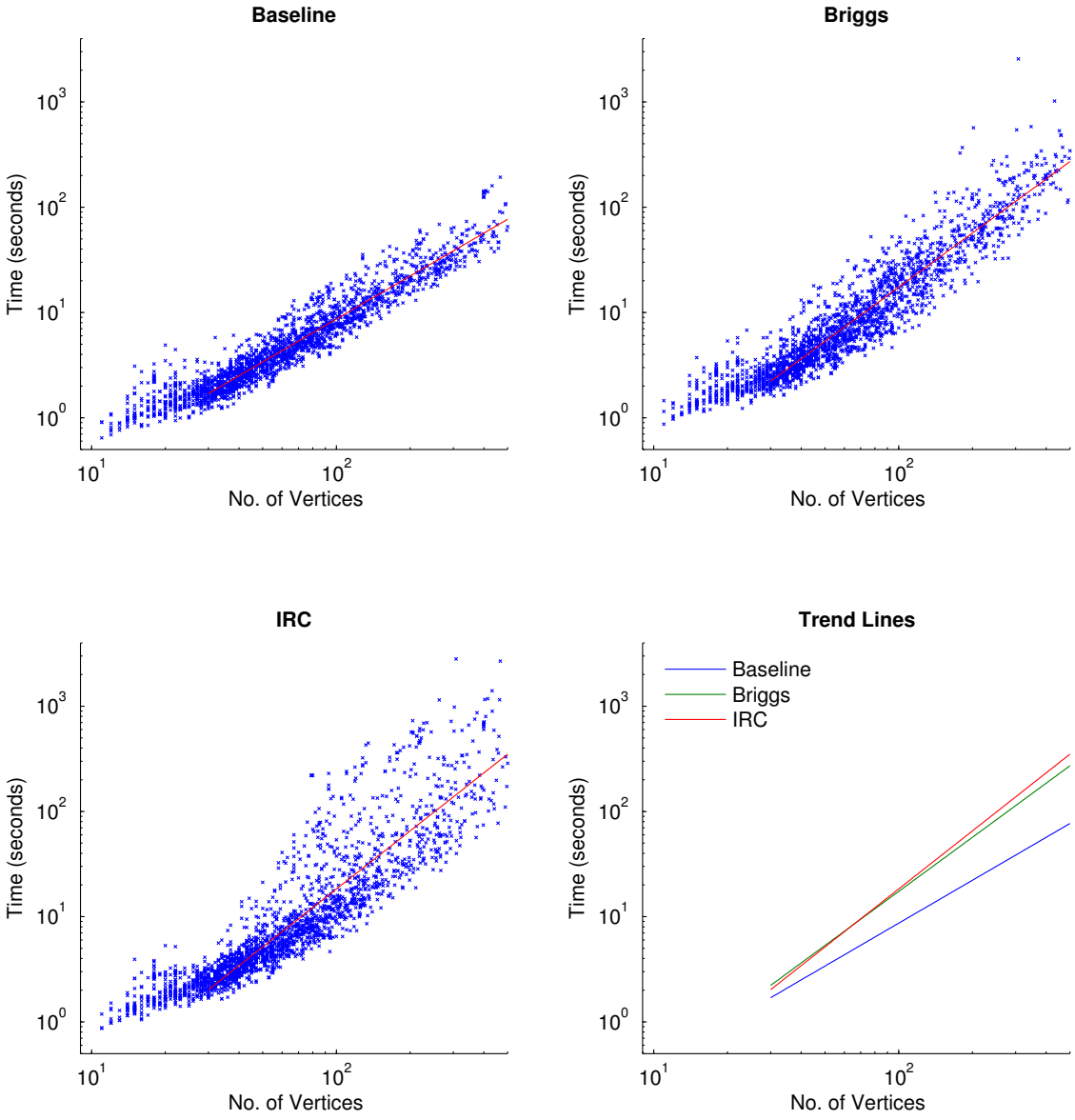

Figure 4.4: Log/Log-scale scatter plots of the time taken (Small Problems).

1. The trend lines show that Baseline is much faster than the other two heuristics for larger problems.

2. The gap between Briggs and IRC is smaller than expected (especially near the start of the graph where IRC is faster). This might be because most of the problems already contain many coalesceable moves.

3. The time taken by Briggs and IRC make them unusable for larger problems; the long run times are due to symbolic evaluation in the theorem prover.

Table 4.1 summarises the average coalescing performance across all the problems. We see that the coalescing performance increases as expected across the heuristics. In particular, implementing IRC was a worthwhile endeavour in this project.

Heuristic	Moves Coalesced (Avg. %)	Improvement Ratio (with respect to Baseline)
Baseline	70.31	1.0
Briggs	88.49	1.26
IRC	92.24	1.31

Table 4.1: Coalescing Performance (Small Problems).

Table 4.2 summarises the spilling behaviour of the heuristics; spills occur in very few cases but IRC appears to spill slightly less vertices on average. The spilling behaviour is otherwise, roughly similar across the heuristics since they all rely on optimistic spilling.

Heuristic	No. of problems with spills	Spilled Vertices (Avg. % in problems with spills)
Baseline	34	1.94
Briggs	31	2.09
IRC	30	1.36

Table 4.2: Spilling Behaviour (Small Problems).

4.4.3 HOL4 Evaluation (Small, Spilling Problems)

To further investigate the spilling behaviour, the same tests were performed with a drastically reduced number of registers for colouring[5].

Figure 4.5 shows the same Log/Log-scale scatter plots on this set. The desired distinction between Briggs and IRC runtimes is more obvious here as there are less opportunities for coalescing.

Table 4.3 shows that the coalescing improvements over Baseline are slightly lowered; IRC is still the leading heuristic.

The spilling behaviour is summarised in Table 4.4. The IRC performs marginally better than the other two heuristics.

[5]The original problems typically had $k = 21$, I used $k = 6$ for all the spilling tests.

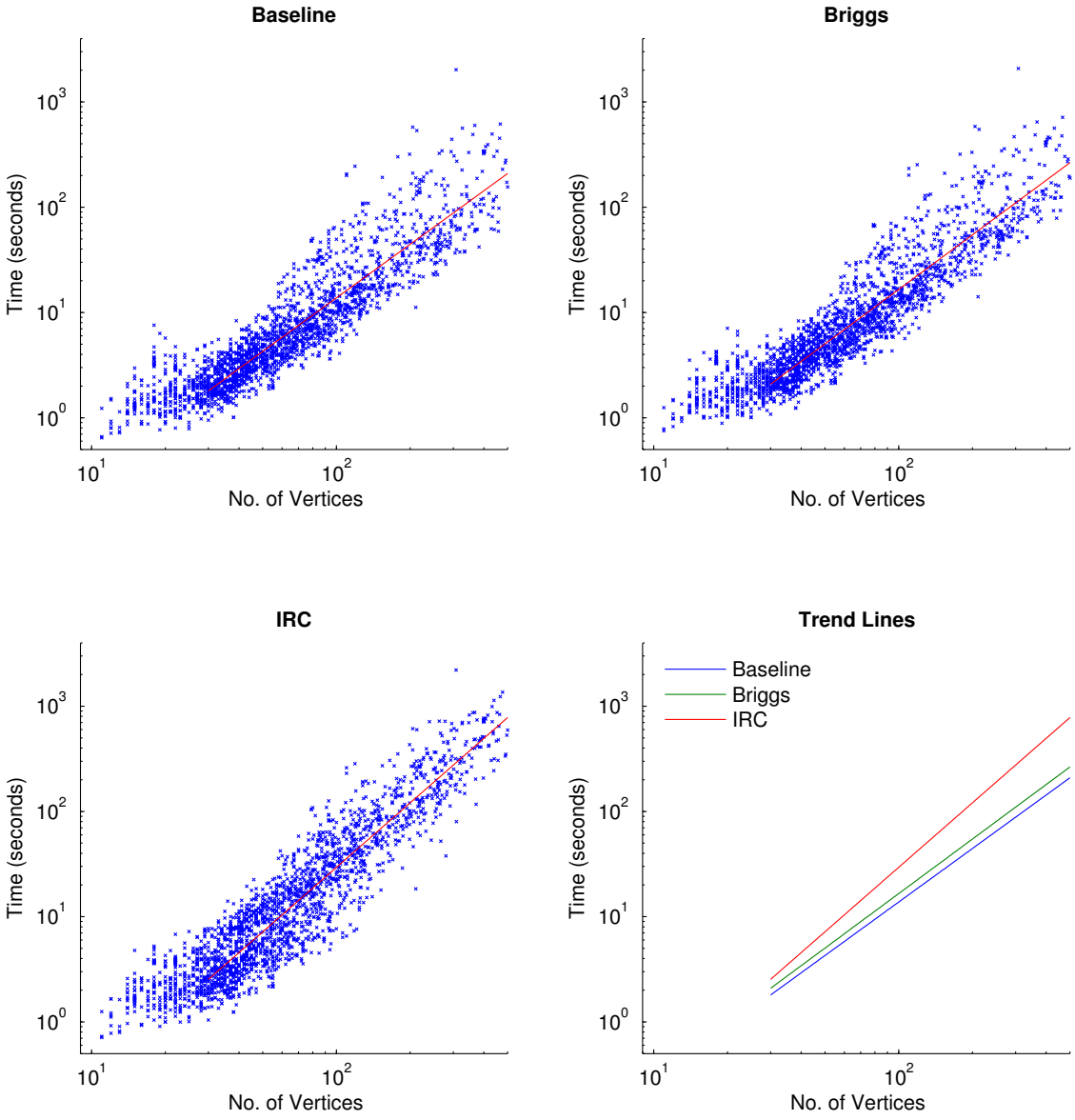

Figure 4.5: Log/Log-scale scatter plots of the time taken (Small, Spilling Problems).

Heuristic	Moves Coalesced (Avg. %)	Improvement Ratio (with respect to Baseline)
Baseline	45.90	1.0
Briggs	47.99	1.05
IRC	54.15	1.18

Table 4.3: Coalescing Performance (Small, Spilling Problems).

Heuristic	No. of problems with spills	Spilled Vertices (Avg. % in problems with spills)
Baseline	1946	12.50
Briggs	1946	12.50
IRC	1934	12.48

Table 4.4: Spilling Behaviour (Small, Spilling Problems).

4.4.4 Poly/ML Evaluation (Large Problems)

Although the CakeML compiler is usually evaluated in HOL4, it is useful to evaluate the translated (unverified) register allocator for two reasons:

1. To examine the performance of the allocator on larger problems.
2. To get an indication of the performance of the allocator when it is bootstrapped (or when it is used as an external, unverified allocator).

For brevity, the Briggs heuristic was excluded since we no longer need the tradeoff it provides. This is evidenced by Figure 4.6 which shows the time taken in Poly/ML for the large problems[6]. There is still a distinction in time between the two heuristics but it is fairly obvious that the IRC algorithm can now be used for all the problems. The longest time taken on a single problem in this large set was merely 401 seconds compared to 2,812 seconds for HOL4 evaluation on the small set.

The coalescing and spilling performance for the large set is shown in Tables 4.5 and 4.6 respectively. Additionally, the outputs obtained for the small set in Poly/ML match those obtained from before in HOL4 using the verified allocator. Hence, we can be reasonably confident that nothing went wrong with the unverified pretty printing and execution in Poly/ML.

Heuristic	Moves Coalesced (Avg. %)	Improvement Ratio (with respect to Baseline)
Baseline	63.22	1.0
IRC	86.22	1.36

Table 4.5: Coalescing Performance (Large Problems).

[6]The small problems all took less than a second for IRC and negligible time for Baseline so they have been excluded from this plot.

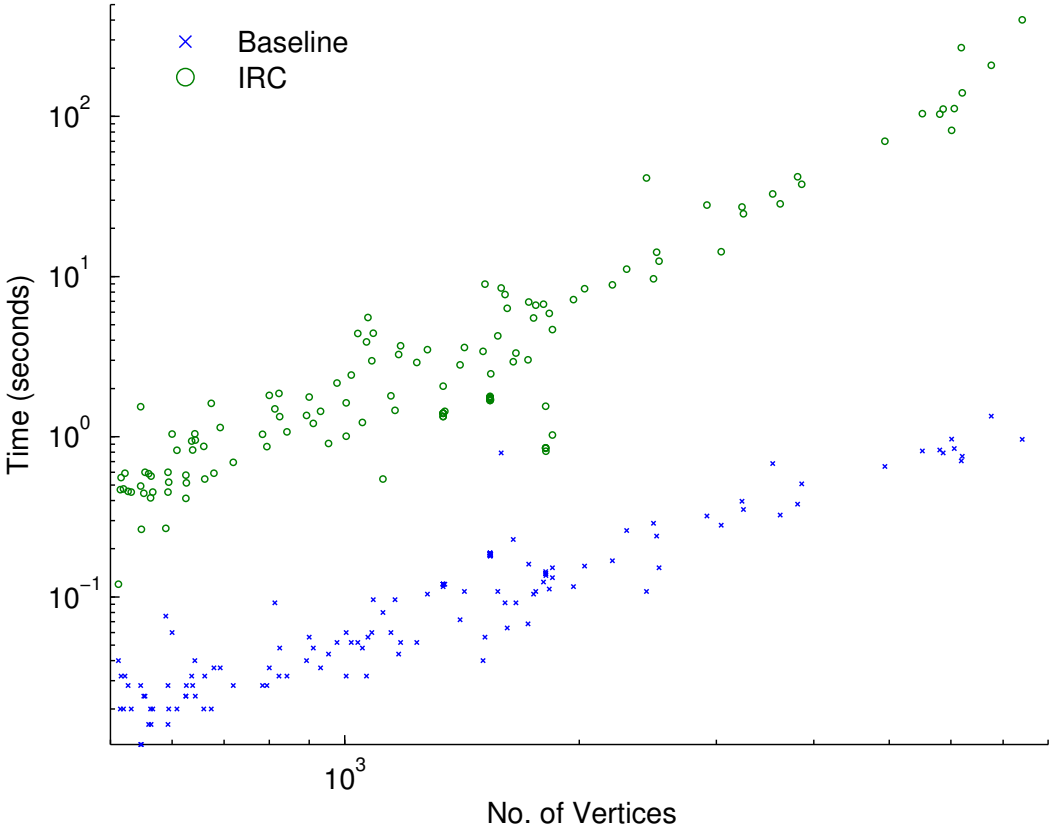

Figure 4.6: Log/Log-scale scatter plots of the time taken (Large Problems).

Heuristic	No. of problems with spills	Spilled Vertices (Avg. % in problems with spills)
Baseline	3	0.86
IRC	3	0.85

Table 4.6: Spilling Behaviour (Large Problems).

4.4.5 Further Analysis

Although IRC has the best coalescing performance, its HOL4 implementation takes too long to be feasibly used in our compiler for larger problems. Two potential solutions are:

1. Use the translation validation approach and the unverified implementation of the register allocator running in Poly/ML. The evaluation results show that this allows us to solve all the problems in a reasonable amount of time.

2. Adaptively switch between the IRC, Briggs and Baseline heuristics using some thresholds defined in the register allocation pass itself. For example, an easily computed adaptation threshold is to switch to Baseline if the number of vertices is greater than a pre-defined value.

The eventual choice of solution in CakeML will depend on the properties of clash graphs that our new backend generates. We also need to make a trade-off between having more speed or remaining in a verified environment.

4.5 Code Size and Build Times

The total code/proof size for this project is about 8000 lines of code, excluding comments and blank lines[7]. It takes roughly 10.5 minutes to do a full build of the theories. This is reasonable with respect to the CakeML compiler build process since the current compiler proofs take about 25 minutes (and the x86 build takes roughly an hour)[8]. A breakdown of the files involved in this project is given in Table 4.7.

[7]Not including word_lemmasScript and word_langScript which were defined and proved before the project. I made minor changes to both files as well.

[8]We are not usually concerned about the build times for CakeML compiler theories.

Script File	Code/Proof Size (Lines of code excluding comments and blank lines)	Build Time (s)
word_langScript (Semantics Definitions)	616	1m 39s
word_lemmasScript (Useful lemmas for handling `Call` and `Alloc` cases)	893	32s
word_procScript (Shared Definitions)	325	11s
word_liveScript (Liveness Analysis)	1653	2m 27s
reg_allocScript (Graph Analysis & Graph Colouring)	2349	1m 23s
word_ssaScript (SSA/CC Pass)	3148	6m 29s
word_transformScript (Final Theorems)	208	3.6s
reg_alloc_translateScript (Translation of the Register Allocator)	149	N.A.

Table 4.7: Code/Proof Size and Build Times for various proof scripts in this project.

Chapter 5

Conclusions

In this dissertation, I have given an explanation of the textbook algorithms used for register allocation, how they were modified in my implementation and how they were verified in the context of the CakeML compiler. This project was an opportunity to contribute to the CakeML compiler and it has certainly been a pleasure working with the other developers.

5.1 Completed Project Goals

The original proposed project goals of writing a verified register allocator for the CakeML backend have been achieved. All of the theorems described in this dissertation are fully verified, i.e. their proofs contain no cheats[1]. The other project success criteria have been qualitatively and quantitatively evaluated in Chapter 4.

I went beyond the original goals by writing an additional SSA/CC pass and implementing the more complex IRC algorithm. These actually took up nearly half the time spent on this project, but they will both be beneficial to the new backend.

5.2 Lessons Learnt

I have gained deeper insights into the interaction between several areas of Computer Science, especially Compilers and Theorem Proving. Formally verifying compiler optimisations forces us to be careful and precise with their definitions. The most significant example was the distinction between live sets and clash sets defined in §2.3; the proof of Liveness Analysis simply would not work with live sets alone!

The experience gained in interactive theorem proving was valuable, as it is not currently taught in the undergraduate part of this course. In particular, it was interesting (and perhaps unsurprising) that software engineering principles are readily applicable to proofs. This project contains much larger-scale proofs than the ones I had previously written for CakeML so it was a great opportunity to apply proof engineering techniques.

[1] It is possible to temporarily skip parts of a proof using the `cheat` tactic.

5.3 Ongoing and Future Work

The new CakeML compiler backend is undergoing active development; my project has been merged into the development branch for further integration and we hope to publish our work in POPL'16.

There are also several other dimensions along which this project can be extended:

- The wordLang semantics could be changed to support a concept of basic blocks and other compiler optimisations. The necessary modifications to this project would be confined to Liveness Analysis because of the proof decoupling.
- Correctness of the IRC algorithm can be fully verified.
- This project has focused on graph colouring based register allocation but it is possible that other types of register allocation heuristics, such as Linear Scan Register Allocation [15], can be implemented and verified as well.
- The proof producing translation mechanism could be extended to automatically translate purely functional (monadic) definitions into stateful CakeML code. This could potentially allow the register allocator to run much faster when it is bootstrapped.

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, New York, NY, USA, 2004.

[2] Andrew W. Appel and Lal George. Sample graph coloring problems. `https://www.cs.princeton.edu/~appel/graphdata/`, 1996. Accessed: 2015-03-12.

[3] Various Authors. CakeML. `https://cakeml.org`, 2015. Accessed: 2015-03-12.

[4] Various Authors. HOL4 Kananaskis-10. `http://hol.sourceforge.net/`, 2015. Accessed: 2015-05-02.

[5] Various Authors. Standard ML of New Jersey. `www.smlnj.org`, 2015. Accessed: 2015-03-16.

[6] D.J. Barker. Developing a formally verified algorithm for static register allocation. Computer Science Tripos Part III Dissertation. University of Cambridge, Computer Laboratory, 2014.

[7] Sandrine Blazy, Benot Robillard, and Andrew W. Appel. Formal Verification of Coalescing Graph-Coloring Register Allocation. In Andrew D. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer Berlin Heidelberg, 2010.

[8] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register Allocation: What Does the NP-completeness Proof of Chaitin Et Al. Really Prove? Or Revisiting Register Allocation: Why and How. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 283–298, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Houston, TX, USA, 1992. UMI Order No. GAX92-34388.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.

[11] Gregory Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Not.*, 39(4):66–74, April 2004.

[12] Lal George and Andrew W. Appel. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996.

[13] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. *SIGPLAN Not.*, 49(1):179–191, January 2014.

[14] Magnus O. Myreen and Scott Owens. Proof-producing Synthesis of ML from Higher-order Logic. *SIGPLAN Not.*, 47(9):115–126, September 2012.

[15] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999.

[16] Silvain Rideau and Xavier Leroy. Validating Register Allocation and Spilling. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 224–243. Springer Berlin Heidelberg, 2010.

[17] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, August 1984.

Appendix A

Definition of wordLang

This appendix gives the full HOL4 definitions of wordLang syntax and semantics.

A.1 Syntax of wordLang

The syntax of wordLang is defined using recursive datatypes in HOL4. Intuitively, these datatypes represent an AST of the program.

Definition A.1.1. Abstract Syntax Tree of wordLang expressions

```
α word_exp =
    Const (α word)
  | Var num
  | Lookup store_name
  | Load (α word_exp)
  | Op binop (α word_exp list)
  | Shift shift (α word_exp) (α num_exp)
```

Definition A.1.2. Abstract Syntax Tree of wordLang programs

```
α word_prog =
    Skip
  | Move num ((num, num) alist)
  | Inst (α inst)
  | Assign num (α word_exp)
  | Get num store_name
  | Set store_name (α word_exp)
  | Store (α word_exp) num
  | Call ((num × num_set × α word_prog × num × num) option)
         (num option) (num list)
         ((num × α word_prog × num × num) option)
  | Seq (α word_prog) (α word_prog)
  | If cmp num (α reg_imm) (α word_prog) (α word_prog)
  | Alloc num num_set
  | Raise num
  | Return num num
  | Tick
```

A.2 Semantics of wordLang

The evaluation semantics of wordLang is defined using an evaluation function. The evaluation of programs is done with respect to a wordLang state.

Definition A.2.1. wordLang State Definition

```
α word_state =
  <| locals : (α word_loc num_map);
     store : (store_name ↦ α word_loc);
     stack : (α word_st list);
     memory : (α word -> α word_loc);
     mdomain : (α word -> bool);
     permute : (num -> num -> num);
     gc_fun : (α gc_fun_type);
     handler : num;
     clock : num;
     code : ((num × α word_prog × num) num_map);
     output : tvarN |>
```

Definition A.2.2. Full definition of `wEval`

```
wEval (Skip,s) = (NONE,s)
wEval (Alloc n names,s) =
case get_var n s of
  SOME (Word w) ⇒ wAlloc w names s
| _ ⇒ (SOME Error,s)
wEval (Move pri moves,s) =
if ALL_DISTINCT (MAP FST moves) then
  case get_vars (MAP SND moves) s of
    NONE ⇒ (SOME Error,s)
  | SOME vs ⇒ (NONE,set_vars (MAP FST moves) vs s)
else (SOME Error,s)
wEval (Inst i,s) =
case wInst i s of NONE ⇒ (SOME Error,s) | SOME s1 ⇒ (NONE,s1)
wEval (Assign v exp,s) =
case word_exp s exp of
  NONE ⇒ (SOME Error,s)
| SOME w ⇒ (NONE,set_var v (Word w) s)
wEval (Get v name,s) =
case FLOOKUP s.store name of
  NONE ⇒ (SOME Error,s)
| SOME x ⇒ (NONE,set_var v x s)
wEval (Set v exp,s) =
case word_exp s exp of
  NONE ⇒ (SOME Error,s)
| SOME w ⇒ (NONE,set_store v (Word w) s)
wEval (Store exp v,s) =
```

```
case word_exp s exp of
  NONE ⇒ (SOME Error,s)
| SOME a ⇒
    case get_var v s of
      NONE ⇒ (SOME Error,s)
    | SOME w ⇒
        case mem_store a w s of
          NONE ⇒ (SOME Error,s)
        | SOME s1 ⇒ (NONE,s1)
wEval (Tick,s) =
if s.clock = 0 then (SOME TimeOut,call_env [] s with stack := [])
else (NONE,dec_clock s)
wEval (Seq c1 c2,s) =
(let (res,s1) = wEval (c1,s)
 in
   if res = NONE then wEval (c2,s1) else (res,s1))
wEval (Return n m,s) =
case get_var n s of
  NONE ⇒ (SOME Error,s)
| SOME x ⇒
    case get_var m s of
      NONE ⇒ (SOME Error,s)
    | SOME y ⇒ (SOME (Result x y),call_env [] s)
wEval (Raise n,s) =
case get_var n s of
  NONE ⇒ (SOME Error,s)
| SOME w ⇒
    case jump_exc s of
      NONE ⇒ (SOME Error,s)
    | SOME (s',l1,l2) ⇒ (SOME (Exception (Loc l1 l2) w),s')
wEval (If cmp r1 ri c1 c2,s) =
case get_var r1 s of
  SOME (Word x) ⇒
    (case get_var_imm ri s of
       SOME (Word y) ⇒
         if word_cmp cmp x y then wEval (c1,s) else wEval (c2,s)
     | _ ⇒ (SOME Error,s))
| _ ⇒ (SOME Error,s)
wEval (Call ret dest args handler,s) =
case get_vars args s of
  NONE ⇒ (SOME Error,s)
| SOME xs ⇒
    case find_code dest xs s.code of
      NONE ⇒ (SOME Error,s)
    | SOME (args1,prog) ⇒
        case ret of
          NONE ⇒
```

```
    if handler = NONE then
      if s.clock = 0 then
        (SOME TimeOut,call_env [] s with stack := [])
      else
        case wEval (prog,call_env args1 (dec_clock s)) of
          (NONE,s) ⇒ (SOME Error,s)
        | (SOME res,s) ⇒ (SOME res,s)
    else (SOME Error,s)
| SOME (n,names,ret_handler,l1,l2) ⇒
    case cut_env names s.locals of
      NONE ⇒ (SOME Error,s)
    | SOME env ⇒
        if s.clock = 0 then
          (SOME TimeOut,call_env [] s with stack := [])
        else
          case
            wEval
              (prog,
               call_env (Loc l1 l2::args1)
                 (push_env env handler (dec_clock s)))
          of
            (NONE,s2) ⇒ (SOME Error,s2)
          | (SOME (Result x y),s2) ⇒
              if x ≠ Loc l1 l2 then (SOME Error,s2)
              else
                (case pop_env s2 of
                   NONE ⇒ (SOME Error,s2)
                 | SOME s1 ⇒
                     if domain s1.locals = domain env then
                       case
                         wEval (ret_handler,set_var n y s1)
                       of
                         (NONE,s) ⇒ (NONE,s)
                       | (SOME v3,s) ⇒ (SOME Error,s)
                     else (SOME Error,s1))
          | (SOME (Exception x' y'),s2) ⇒
              (case handler of
                 NONE ⇒ (SOME (Exception x' y'),s2)
               | SOME (n,h,l1,l2) ⇒
                   if x' ≠ Loc l1 l2 then (SOME Error,s2)
                   else if domain s2.locals = domain env then
                     wEval (h,set_var n y' s2)
                   else (SOME Error,s2))
          | (SOME TimeOut,s2) ⇒ (SOME TimeOut,s2)
          | (SOME NotEnoughSpace,s2) ⇒
              (SOME NotEnoughSpace,s2)
          | (SOME Error,s2) ⇒ (SOME Error,s2)
```

Appendix B

Proof of Liveness Analysis Correctness

This appendix gives an overview of my proof process for Theorem 3.2.4. Note that it is meant for readers unfamiliar with interactive proof and will consist of a series of screenshots taken during various stages of the interactive proof process.

B.1 Setting up the proof

```
val it = (): unit
> Vim input /tmp/vimhol0
val it =
   Initial goal:



∀prog st cst f live.
  colouring_ok f prog live ∧ word_state_eq_rel st cst ∧
  strong_locals_rel f (domain (get_live prog live)) st.locals
    cst.locals ⇒
  ∃perm'.
    (let (res,rst) = wEval (prog,st with permute := perm')
     in
       if res = SOME Error then T
       else
         (let (res',rcst) = wEval (apply_colour f prog,cst)
          in
            res = res' ∧ word_state_eq_rel rst rcst ∧
            case res of
              NONE =>
                strong_locals_rel f (domain live) rst.locals rcst.locals
            | SOME v1 => T))
:
   proof
```

Figure B.1: The interactive proof is started by formally stating our top level goal in the HOL4 proof manager.

```
              case res of
                NONE =>
                  strong_locals_rel f' (domain live') rst.locals
                    rcst.locals
              | SOME v1 => T))
  1.  v = word_prog_size (K 0) Skip
  2.  colouring_ok f Skip live
  3.  word_state_eq_rel st cst
  4.  strong_locals_rel f (domain (get_live Skip live)) st.locals
        cst.locals
------------------------------------
∃perm'.
  (let (res,rst) = wEval (Skip,st with permute := perm')
   in
     res = SOME Error ∨
     (let (res',rcst) = wEval (apply_colour f Skip,cst)
      in
        res = res' ∧ word_state_eq_rel rst rcst ∧
        case res of
          NONE =>
            strong_locals_rel f (domain live) rst.locals rcst.locals
        | SOME v1 => T))

14 subgoals
:
   proof
```

Figure B.2: An induction tactic is applied to start a proof by structural induction. Notice that we have a subgoal for each of the 14 syntactic constructors of wordLang.

B.2 Proving `Move`

```
              case res of
                NONE =>
                  strong_locals_rel f' (domain live') rst.locals
                    rcst.locals
              | SOME v1 => T))
  1.  v = word_prog_size (K 0) (Move n l)
  2.  colouring_ok f (Move n l) live
  3.  word_state_eq_rel st cst
  4.  strong_locals_rel f (domain (get_live (Move n l) live)) st.locals
        cst.locals
------------------------------------
∃perm'.
  (let (res,rst) = wEval (Move n l,st with permute := perm')
   in
     res = SOME Error ∨
     (let (res',rcst) = wEval (apply_colour f (Move n l),cst)
      in
        res = res' ∧ word_state_eq_rel rst rcst ∧
        case res of
          NONE =>
            strong_locals_rel f (domain live) rst.locals rcst.locals
        | SOME v1 => T))

13 subgoals
:
   proof
```

Figure B.3: Initial subgoal corresponding to `Move` commands. The statements printed above the dashed line are our **assumptions**.

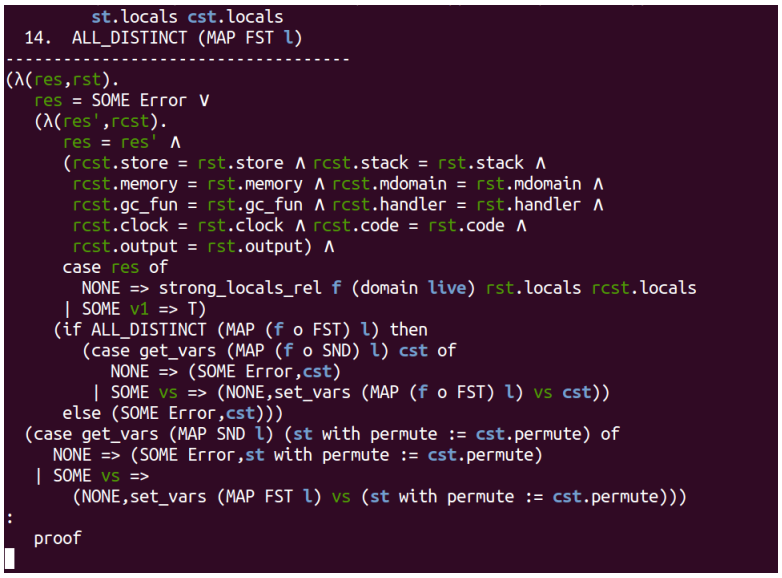

Figure B.4: The evaluation semantics is expanded using rewrite and simplification tactics. We find that the original destination registers, `MAP FST` l, must be `ALL_DISTINCT`. This is automatically added to our assumptions list (Assumption 14).

Figure B.5: By the construction of clash sets, we know that f is injective over the original destination registers. Therefore the coloured destination registers, `MAP` f `(MAP FST` l`)`, are also `ALL_DISTINCT`. This is manually proven as a new sub-goal and added to the assumptions (Assumption 15).

```
          (wEval (prog',st' with permute := perm'))
  1.  v = word_prog_size (K 0) (Move n l)
  2.  INJ f (domain (FOLDR delete live (MAP FST l)) ∪ set (MAP SND l))
        𝕌(:num)
  3.  INJ f (set (MAP FST l) ∪ domain live) 𝕌(:num)
  4.  cst.store = st.store
  5.  cst.stack = st.stack
  6.  cst.memory = st.memory
  7.  cst.mdomain = st.mdomain
  8.  cst.gc_fun = st.gc_fun
  9.  cst.handler = st.handler
  10.  cst.clock = st.clock
  11.  cst.code = st.code
  12.  cst.output = st.output
  13.  strong_locals_rel f
         (domain (FOLDR delete live (MAP FST l)) ∪ set (MAP SND l))
         st.locals cst.locals
  14.  ALL_DISTINCT (MAP FST l)
  15.  ALL_DISTINCT (MAP (f o FST) l)
  16.  get_vars (MAP SND l) st = SOME x
  17.  get_vars (MAP (f o SND) l) cst = SOME x
------------------------------------
strong_locals_rel f (domain live) (list_insert (MAP FST l) x st.locals)
  (list_insert (MAP (f o FST) l) x cst.locals)
:
   proof
```

Figure B.6: Assumption 15 allows us to further simplify the goal. It remains to show that `strong_locals_rel` continues to hold between the two local states after the command has been executed.

```
            NONE => (SOME Error,cst)
          | SOME vs => (NONE,set_vars (MAP (f o FST) l) vs cst))
       else (SOME Error,cst)))
    (case get_vars (MAP SND l) (st with permute := cst.permute) of
       NONE => (SOME Error,st with permute := cst.permute)
     | SOME vs =>
         (NONE,
          set_vars (MAP FST l) vs (st with permute := cst.permute)))

Goal proved.
 [.....]
|- ∃perm'.
     (let (res,rst) = wEval (Move n l,st with permute := perm')
      in
        res = SOME Error ∨
        (let (res',rcst) = wEval (apply_colour f (Move n l),cst)
         in
           res = res' ∧ word_state_eq_rel rst rcst ∧
           case res of
             NONE =>
               strong_locals_rel f (domain live) rst.locals rcst.locals
           | SOME v1 => T))

Remaining subgoals:
val it =
   ...2 subgoals elided...
```

Figure B.7: The proof is finished by case analysis on the definition of `strong_locals_rel`.

B.3 Proving Seq

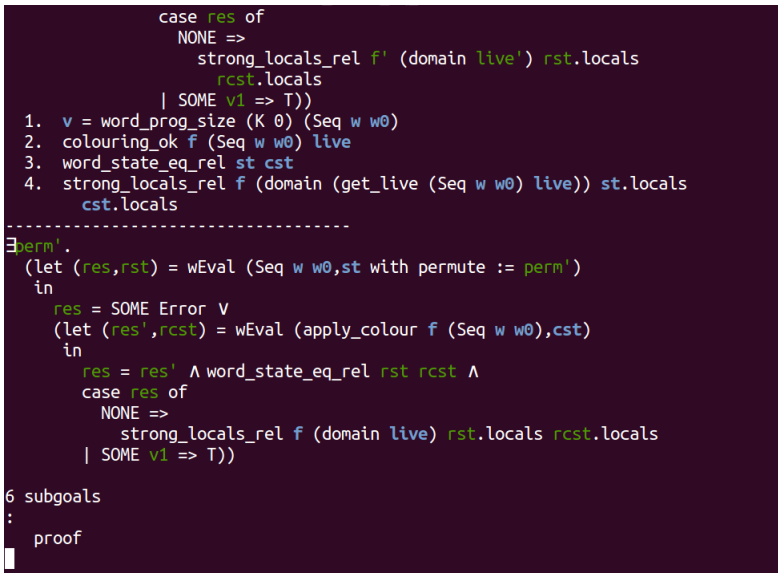

Figure B.8: Initial subgoal corresponding to `Seq` commands.

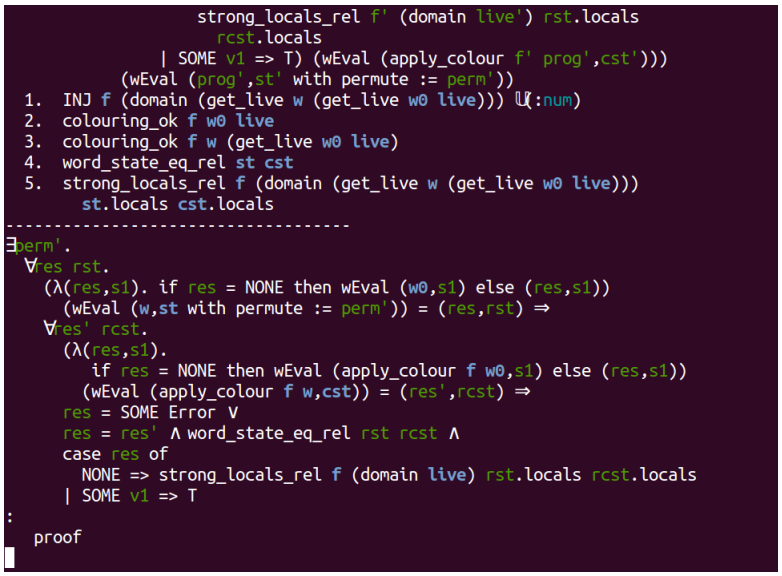

Figure B.9: By expanding the semantics, we find that we need to use the inductive hypothesis on the first part of the sequence (w).

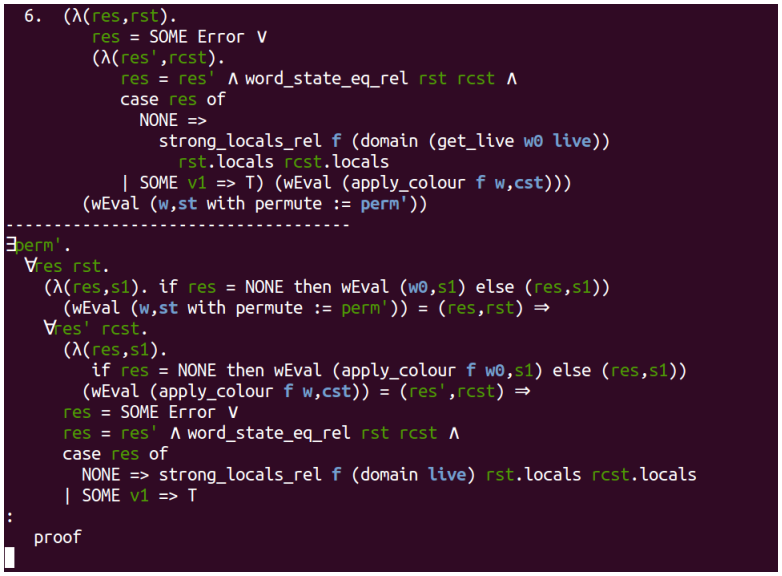

Figure B.10: The inductive hypothesis for w is instantiated from Assumption 0 (not shown here), yielding Assumption 6.

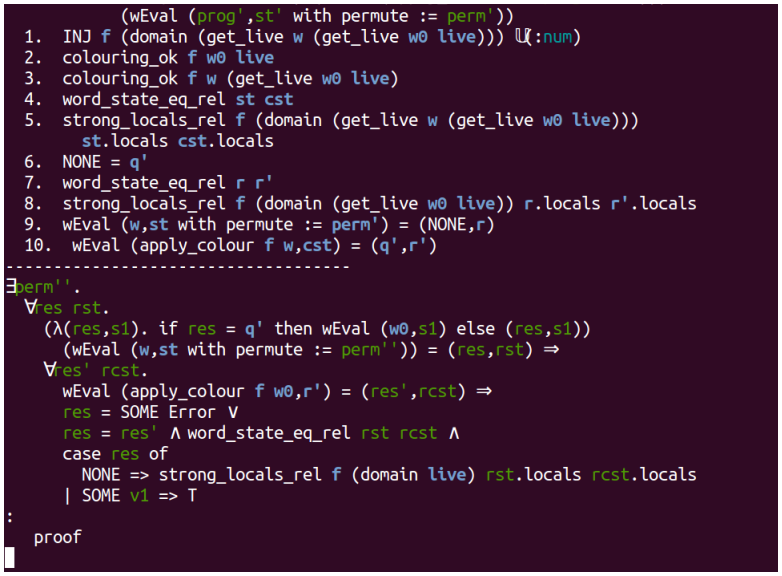

Figure B.11: Recall that `Seq` does not continue evaluation unless w evaluates to `NONE`. We do a case split and trivially prove the other case. Now, we are left with the case where evaluating w gave us `NONE` (Assumption 9).

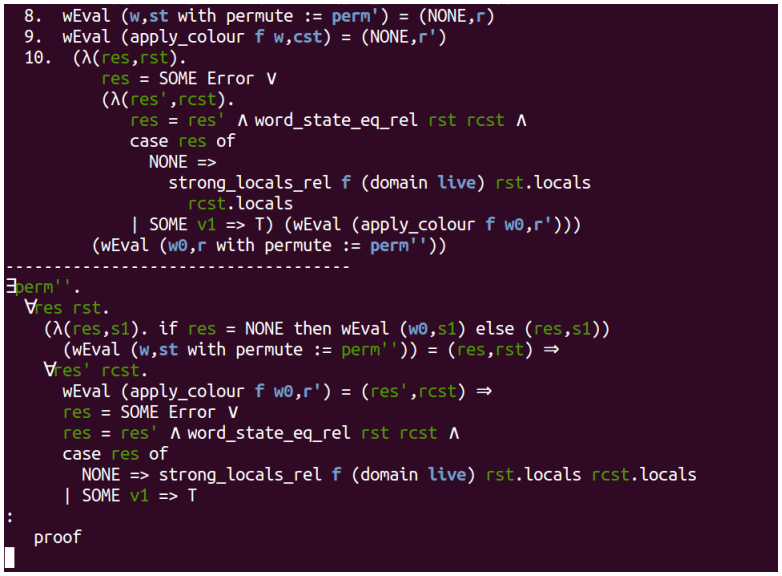

Figure B.12: Naturally, we use the inductive hypothesis again on the second part of the sequence (u_0). We are almost done but we have a mismatch in the permutation oracles: Assumption 8 concludes with arbitrary r but Assumption 10 requires r `with permute :=` $perm''$. Note that Assumption 8 is just a renumbering of Assumption 9 from the previous figure.

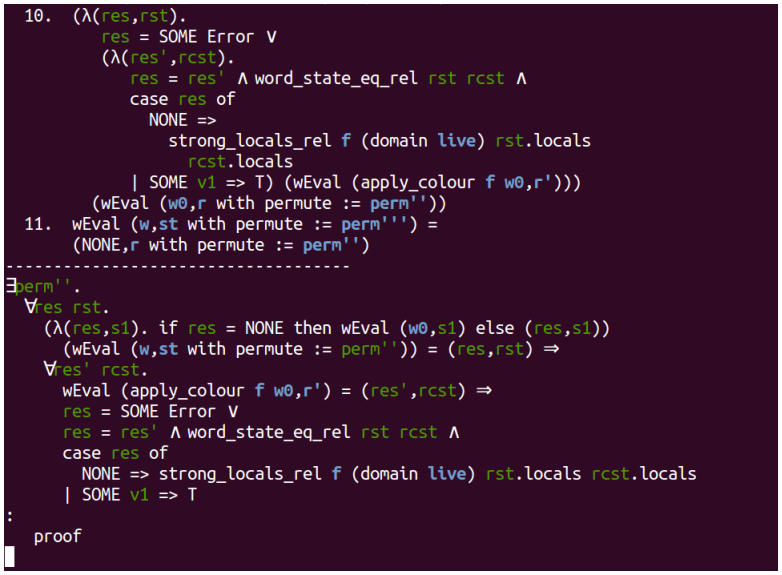

Figure B.13: To fix the mismatch, we instantiate the permutation oracle swapping theorem on Assumption 8 to get Assumption 11.

```
        (res',rcst) ⇒
        res = SOME Error ∨
        res = res' ∧ word_state_eq_rel rst rcst ∧
        case res of
          NONE =>
            strong_locals_rel f (domain live) rst.locals rcst.locals
        | SOME v1 => T

Goal proved.
 [....]
|- ∃perm'.
     (let (res,rst) = wEval (Seq w w0,st with permute := perm')
      in
        res = SOME Error ∨
        (let (res',rcst) = wEval (apply_colour f (Seq w w0),cst)
         in
           res = res' ∧ word_state_eq_rel rst rcst ∧
           case res of
             NONE =>
               strong_locals_rel f (domain live) rst.locals rcst.locals
           | SOME v1 => T))

Remaining subgoals:
val it =

  0.  ∀prog' st' cst' f' live'.
        word_prog_size (K 0) prog' < word_prog_size (K 0) Tick ⇒
```

Figure B.14: The proof is finished by providing the appropriate witness, *perm'''*.

B.4 Completing the proof

```
+0+0+0+0+0+0+0+0+1#
metis: r[+0+110]+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0-
1+0+0+0+2+2+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+1+0+0+2+0+4+4+0+5+2+0+0+10+0-
0+2+1+1+2+0+0+2+3+3+3+6+0+0+0+3#
metis: r[+0+103]+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0-
0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+1+0+0+0+0+0+0+2+0+0+3+6
val it =
   Initial goal proved.
|- ∀prog st cst f live.
     colouring_ok f prog live ∧ word_state_eq_rel st cst ∧
     strong_locals_rel f (domain (get_live prog live)) st.locals
       cst.locals ⇒
     ∃perm'.
       (let (res,rst) = wEval (prog,st with permute := perm')
        in
          if res = SOME Error then T
          else
            (let (res',rcst) = wEval (apply_colour f prog,cst)
             in
               res = res' ∧ word_state_eq_rel rst rcst ∧
               case res of
                 NONE =>
                   strong_locals_rel f (domain live) rst.locals
                     rcst.locals
               | SOME v1 => T)):
   proof
```

Figure B.15: The full proof is completed by proving the various other subgoals. The sequence of tactics used in the proof is stored in a proof script (word_liveScript.sml) so that we can later reconstruct or modify the proof easily.

Appendix C

Project Proposal

CST Part II Project Proposal

Verified Register Allocation for CakeML

Y. K. Tan, Robinson College

Originators: Dr M. Myreen / R. Kumar

Date: 21^{st} October 2014

Project Supervisor: R. Kumar

Director of Studies: Dr A. Beresford

Project Overseers: Dr A. Rice & Dr T. Griffin

Introduction

CakeML[1] is a verification-friendly programming language with a verified compiler written in the HOL4 Theorem Prover[2].

Compilation of high level source code into machine code is a complex task. It is usually broken down into several phases e.g. Lexing, Parsing, Optimization / Translation and finally Code Generation. The CakeML compiler is similarly broken down into several phases. This project is concerned with the backend optimization phase of the compiler.

During compilation, it is much easier for the compiler to assume the underlying machine has an infinite number of imaginary registers to hold intermediate values. Hence, code like:

```
val x = a+b;
val y = x+c;
val z = y+d;
...
```

might be compiled into an intermediate assembly-like language:

```
ADD r0 a b
ADD r1 r0 c
ADD r2 r1 d
...
```

Real machines have a finite number of registers so, naively, any imaginary registers (e.g. r32 and above) would be spilled[3]. This is unsatisfactory as reading/writing a variable from memory typically requires far more instructions and CPU cycles than one already in a physical register (even with caching).

Register allocation alleviates this problem by allocating imaginary registers to physical registers in a way that tries to minimize spilling. In the above example, if we know the values of x and y are never used again, then it is sufficient to have a single temporary holding their values when calculating z:

```
ADD r0 a b
ADD r0 r0 c
ADD r0 r0 d
...
```

Liveness analysis determines the variables that must be simultaneously live (and therefore, must occupy distinct register/stack locations) at each instruction. The resulting live-sets are used to perform safe register allocation.

CakeML's backend is organized into several intermediate languages with operational semantics defined in HOL4. This project will target one such language for register allocation. The primary task is to write the above passes and prove that the resulting

[1] `https://cakeml.org`

[2] `http://hol.sourceforge.net`

[3] Stored in stack locations and reads or writes to these registers are eventually converted to their memory load/store equivalents

allocations preserve the semantics of input programs. Various other passes may also be written to setup the liveness analysis / improve performance of register allocation.

Starting Point

I did a 10 week research internship with the CakeML developers so I am already familiar with:

1. Interaction with the HOL4 theorem prover (in Vim/Standard ML)
2. Work related to CakeML's compiler, including:
   (a) CakeML Compiler Explorer[4]
   (b) Type Inferencer completeness (partial work)
3. Semantics of the Intermediate Language (IL) being targeted for register allocation

Parts of this project may extend previous work:

1. Verified transformation/calling convention passes on the IL (written by myself)
2. Verified liveness analysis and register allocation for straight-line Three-address code (written by David J. Barker for his Part III dissertation)

Resources Required

I will be using my own laptop (8GB RAM, Ubuntu 14.04). The required software (PolyML, HOL4, CakeML, Lem) is freely available online. Code sources will be committed onto a branch of CakeML on Github which I already have access to. Dissertation sources will similarly be kept on Github in a private repository.

Overview of Technical Work

The project has several sub-tasks, each of which can be further divided into: 1) Writing an implementation and 2) Proving the implementation correct with respect to the semantics. Note that all of these passes will be written as IL to IL transforms so they will be based on the same underlying semantics. All definitions and proofs will be written in the HOL4 Theorem Prover (and Standard ML).

1. Liveness Analysis - Input programs would have to undergo an initial pass that annotates them with live-sets at each instruction. David's work contains an example of liveness analysis for straight line code. However, the IL here has a much richer set of instructions (e.g. Function calls, Allocation, If-Then-Else) and semantic features such as a Garbage Collector. A correct implementation would require careful study of the semantics.

[4] `https://cakeml.org/explorer.cgi`

2. Register Allocation - The standard approach to register allocation is via transformation to a graph coloring problem. Graph coloring is NP-complete in general so practical allocators are approximate algorithms. David's thesis provides several heuristics which will be studied and extended/reimplemented where necessary.

3. Correctness Theorem for Liveness Analysis and Register Allocation - Linking up Liveness Analysis and Register Allocation directly is difficult. A good way to modularize this proof is to define an invariant on colorings / live-sets and prove that:

   (a) Colorings/Allocations respecting this invariant do not change the semantics of input programs

   (b) The register allocation algorithm always produces a coloring satisfying this invariant[5]

   This approach would also allow future work to replace the register allocator cleanly without having to re-prove liveness analysis.

4. Calling Conventions - Not all program registers can be mapped to physical registers in the IL semantics. For example, local variables are forced onto the stack when a function call is made. A calling convention pass marks the variables so that the register allocator ignores them during allocation. This will extend work I did (listed above) with a more complex calling convention.

Success Criterion

The project would be considered successful if a suitable correctness theorem is stated and proved (possibly with cheats[6]) for each item described above. Suitable correctness theorems should allow these items to be added to the CakeML compiler safely. Evaluation of the project will cover:

1. Discussion on proof progress for each theorem, including informal argument for any major unproved subgoals[7].

2. Guarantees provided by the correctness theorems for individual components.

3. How the correctness theorems for register allocation fit into the CakeML compiler backend proofs.

4. Performance of different register allocation heuristics on sample IL programs.

[5] David's correctness proofs will be extended where necessary

[6] More difficult or tedious subgoals can be skipped by inserting a cheat in the proof. Ideally, none of the proofs produced will contain cheats

[7] This will not be necessary if the theorem is completely proved

Possible Extensions

There is room for further work related to the project:

1. Register allocation performance can be improved by optimization passes before allocation (e.g. SSA pass, live range reduction).
2. Register allocation algorithm can be replaced with a more complex algorithm.
3. Optimizations enabled by register allocation can be verified (e.g. dead code removal).
4. Ensuring that the register allocator works through the HOL-to-CakeML translator (used for bootstrapping the compiler). The translator can also be extended to produce stateful CakeML for a more efficient result.
5. For passes that aim at improving quality of output code, evaluation on sample programs can be performed.

Timetable and Milestones

1. **24/10/2014 - 07/11/2014:** Familiarization with David's liveness analysis and register allocation code. Start writing liveness analysis for the IL.
2. **08/11/2014 - 21/11/2014:** Finish liveness analysis and write the register allocator using its output.
3. **22/11/2014 - 06/12/2014:** State correctness theorem for liveness analysis (Item 3a) and start trying to prove it.
4. **07/12/2014 - 26/12/2014:** Finish correctness proof for liveness analysis.
   This is the first project milestone.
5. **27/12/2014 - 16/01/2015:** State correctness theorem for register allocation (Item 3b) and start trying to prove it.
6. **17/01/2015 - 30/01/2015:** Write progress report and prepare progress report presentation. Continue work on correctness proof for register allocation.
   Progress report deadline: 12PM, 30/01/2015
7. **31/01/2015 - 13/02/2015:** Finish correctness proof for register allocation. Finish linking up correctness theorems.
   This is the second project milestone.
8. **14/02/2015 - 27/02/2015:** Quantitative evaluation of various register allocation heuristics for dissertation's Evaluation chapter.
9. **28/02/2015 - 13/03/2015:** Start writing main dissertation chapters with work finished so far.

10. **14/03/2015 - 03/04/2015:** Work on extensions for the project.

11. **04/04/2015 - 17/04/2015:** Finish first draft of the dissertation.
    First draft deadline: 21/04/2015

12. **18/04/2015 - 01/05/2015:** Tie up loose ends in project found when writing up, finish any remaining work on extensions. Continue work on dissertation.

13. **02/05/2015 - 15/05/2015:** Proof reading and submission of dissertation.
    Dissertation submission deadline: 12PM, 15/05/2015