Developing a formally verified algorithm for static register allocation

David J. Barker
Jesus College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for Part III of the Computer Science Tripos

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: djb218@cam.ac.uk

May 29, 2014

Declaration

I David J. Barker of Jesus College, being a candidate for the Part III in Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: TBC

Signed:

Date:

Abstract

This dissertation describes the implementation and verification of a full register allocation algorithm, designed to be representative of the sort of allocator that might be used in a practical optimising compiler. A model of three-address code (and its execution behaviour) was developed and a verified live variable analysis function and clash graph generator were constructed. A number of different graph colouring algorithms were then implemented and proved correct, as well as various heuristics. The work was then tied together by an overarching proof of correctness which demonstrates that the algorithm does not change code behaviour. The algorithm was also extended to construct preference graphs based on 'move' instructions and use them in colouring, and to handle spilling of registers where not enough physical registers are available, and these additions were also fully verified.

Contents

List of Figures

Chapter 1

Introduction

Register allocation is one of the most important stages in an optimising compiler. Due to the significant speed difference between CPU registers and lower-level caches, a program which makes efficient use of registers and minimises memory accesses will be considerably faster than one which naïvely stores variables in memory. The responsibility of register allocation is therefore to map program variables into physical registers whilst also minimising the number of values which have to be stored in memory. As with all optimisation steps, special care is needed to ensure register allocation does not accidentally alter program behaviour. Some values may be allocated the same register on the assumption that they will not be using it at the same time, and for the sake of correctness it is critical that this assumption is valid.

The objective of this project was to formally model a realistic register allocation algorithm in the HOL4 theorem prover, and verify that it does not change program behaviour. The approach used was to begin with a very simple graph colouring algorithm and incrementally improve it, adding extra heuristics and making the model more realistic, whilst maintaining a proof of overall correctness with respect to the semantics of code evaluation. Overall, the project was successful in its goals: a full end-to-end algorithm was verified, from source three-address code to output code with registers allocated, which includes a colouring algorithm and clash graph generator

representative of those used in practice. The system is reasonably modular, and extra heuristics can be added and verified with little difficulty. It also models some more difficult aspects of register allocation, namely preference graphs and register spilling, and the algorithms handling these were fully verified as well.

The next few chapters examine the background surrounding the problem of register allocation, and review some recent related work. Chapters 4 through 7 go on to describe in detail how the system was modelled and verified. In chapter 4, a series of functions for performing live-variable analysis and generating the corresponding clash graphs is described, alongside descriptions of their correctness proofs. Chapter 5 goes on to explore the core of the algorithm: taking a clash graph and producing a colouring, through a combination of vertex order heuristics and colouring algorithms. The work of these chapters is brought together in Chapter 6, where the individual correctness proofs are used to build an overall proof of correctness with respect to code evaluation. Finally, Chapter 7 examines some of the extension work which was completed once the main algorithm had been verified.

Chapter 2

Background

2.1 The problem of register allocation

When compiling code, we generally assume the existence of an infinite number of locations in which to store values. Each variable used by the program is treated as its own location, and temporaries and results are freely allocated to conceptual registers. However, in reality all processors only have a finite number of physical registers, some of which are reserved for specific purposes, and so the compiler ultimately has to map variables in the code to registers in such a way that ensures the code's behaviour is the same.

There are many issues involved when doing this. The main problem is that code will often use far more 'virtual' registers than actually exist on the target processor. The simplest solution to this is to store any values which won't fit in memory, but this considerably worsens performance. The solution is to find variables which are never in use at the same time and place them in the same register, allowing registers to be re-used when the values originally assigned to them are not in use.

This optimisation is enabled by liveness analysis. Essentially, a variable is live where changing its value will affect the program's future behaviour. Because this is difficult to decide with absolute precision, we generally compute a safe

over-approximation of liveness: a variable is live if an instruction reachable from here reads its value, and its value is not written to in a preceding instruction. Liveness is computed by repeatedly iterating over instructions in the code from the end backwards, updating each instruction's set of live variables according the following liveness equation:

$$live(n) = (live(n+1) \setminus write(n)) \cup read(n)$$

(Where $live(n)$ is the set of variables live at instruction n, $live(n+1)$ is those live at the next instruction and $read(n)$ and $write(n)$ are the variables read and written by the instruction.)

This is repeated until the live variable sets stop changing. With this information, we can deduce which variables are live at the same time, and thus which variables cannot occupy the same register because doing so would result in one overwriting the other whilst the other was about to be used.

When no allocation can satisfy these clashes between variables, it becomes necessary to spill variables to memory. This means reading or writing said variables takes considerably longer, and so an effective register allocation algorithm should minimise spills wherever possible.

Another thing to consider is that some target architectures have constraints on what registers can be used for. For instance, a particular operator may require its operands to be stored in particular registers, or may place the output in a designated 'result' register. This means move instructions will be required to satisfy these requirements, slowing the overall program down. Good register allocators allow for this by biasing allocation so that these move instructions can be eliminated wherever possible.

2.2 Isomorphism to graph colouring

The most common approach to register allocation is to reduce the problem to one of graph colouring, where we have a graph which we wish to colour with a maximum of K colours such that no two adjacent vertices share the

same colour (Weisstein [14]). Given the knowledge of which variables are live at the same time, we construct a clash graph in which the vertices represent registers and an edge between two vertices exists where the corresponding registers are simultaneously live. Treating a colour as representing a physical register on the target machine, we then attempt to 'colour' the graph with target registers such that vertices which are linked by a clash edge are always given different colours.

For example, consider the code below, where each instruction is annotated with the set of variables live immediately before it:

```
R1 = R2 + R3   {R2,  R3}
R4 = R1 * R2   {R1,  R2,  R3}
R5 = R3 - R4   {R1,  R3,  R4}
R6 = R1 + R5   {R1,  R5}
```

The clash graph for this code is given below:

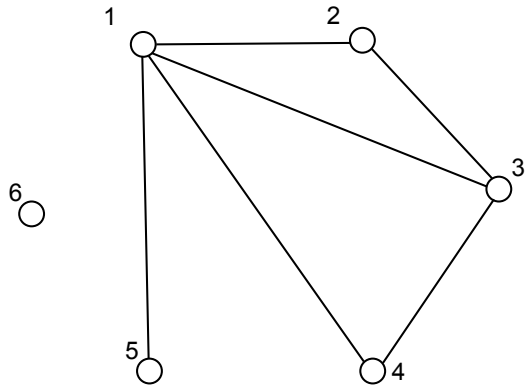

Figure 2.1: Clash graph generated for the sample code

A successful colouring will naturally give a mapping from virtual to physical registers enforcing that no registers which are simultaneously live are placed in the same physical register. Formally verifying that the colouring resulting from this process will never affect the behaviour of the code being compiled was the core objective of this project.

The K-colourability problem is known to be NP-complete wherever K is greater than two, and so finding a truly optimal solution is rarely feasible in

practice (as discussed by Garey & Johnson [4]). Instead, register allocation algorithms are designed primarily to find a solution which is correct, and from there use heuristics to find a solution which is as optimal as can be achieved in reasonable time. There are several approaches to this, and many of them were verified in detail in this project. As discussed later, the definitions and proofs in the project are designed to be largely independent of the algorithm and so extending it to verify some of the algorithms not covered would be relatively simple.

2.3 Basic approaches to colouring

The simplest way of colouring a graph is to take each vertex in turn and assign it a new colour until all vertices have been coloured. This approach is guaranteed to generate a valid colouring (as verified in Section 5.2), but is also very inefficient as registers are never re-used for variables which could reside in the same register. In a system with few physical registers this will result in registers being spilled to memory, and as the order colours are assigned is arbitrary this could result in frequently-used registers being stored in memory causing constant loads and stores and poor performance.

An improvement is to take each vertex in turn and assign it the lowest register which isn't already in use by a vertex with which it clashes. This is fairly simple to implement, and significantly improves register utilisation as it re-uses old registers whenever the opportunity presents itself.

However, lowest-first allocation doesn't always find opportunities to re-use old registers. A further refinement is to check whether we can avoid using a new register by swapping colours of already-allocated vertices. This often improves register utilisation further, as sub-optimal decisions made early on in the process can be changed when colouring new vertices later.

The algorithm resulting from applying all these refinements will work fairly well, but in practice the quality of the colouring is often also heavily dependent on the order in which we colour vertices. Decisions made at the

beginning naturally constrain decisions made later, even if we allow the algorithm to swap existing colours. We therefore need to use a heuristic ordering step at the beginning to improve the order in which vertices are presented to the main colouring algorithm. Some such heuristics are described in the next section.

2.4 Heuristics

Heuristics are used in graph-colouring algorithms to select a vertex ordering which gives a more optimal solution. There are several heuristics which are used in practice, some of which were formally verified during the course of this project.

One of the simplest heuristics is to order the vertices by their degree. By presenting the vertices to the colouring algorithm in descending order of degree, vertices which are most likely to clash with others are coloured first so that those left over towards the end will be easier to colour without assigning new colours. There are also a few variations on this: saturation degree ordering, for instance, picks the next vertex to colour based on which vertex is currently adjacent to the most different colours. Incidence degree colouring is another variant which sorts based on the number of adjacent vertices which have already been coloured (Al-Omari & Sabri [1]).

A more complex heuristic, which is useful when attempting to colour with at most K colours, is to pick at each stage the vertex with the lowest degree (if it is less than K). This vertex is then removed from the graph and placed on top of a vertex stack. The process repeats, each time removing a vertex which is guaranteed to be colourable, until no vertices of degree less than K remain. Those which have not yet been removed are spilled to memory, and the rest of the vertices are popped off the stack one by one and passed to the colouring algorithm. The heuristic essentially priorities registers with few conflicts, spilling those with many conflicts to memory, and colours those of highest degree first.

This is not an exhaustive list of the heuristics used in practice, but these two types are very common and so verification focussed primarily on these. As mentioned, most of the verification conditions were designed to be as general as possible such that other more exotic heuristics could also be verified without much additional effort.

Chapter 3

Related Work

Although some considerable work has gone into the formal verification of programs, and more recently into the compilers themselves, the problem register allocation has not been explored in quite as much depth. However, there are nonetheless several related publications which attempt to formally verify register allocators (and, more generally, graph-colouring algorithms) in various different ways. In this section, a number of these approaches are described and compared, so as to give a background to the work undertaken for this project.

There have been several recent approaches to formalising graph theory in a proof system like HOL. A notable example is Noschinski [10], in which a full library of graph theory primitives is developed for the Isabelle proof system. More relevantly, Bauer & Nipkow [2] specifically discusses formalising graph colouring in Isabelle/HOL as it tackles the problem of verifying the five colour theorem. However, neither of these was found to be general enough to build upon for this project – ultimately, a pair of customised graph representations was used, both of which were designed to best fit their specific applications.

A recent example of a verifying register allocator is the one employed by the CompCert compiler (Leroy [8]), a compiler for a subset of the C language which has been formally verified in Coq. This was proven never to generate

code with behaviour differing from the specified behaviour of the input code, and featured a graph-colouring register allocator whose output was checked by a formally verified algorithm designed to ensure the colouring was correct (Rideau & Leroy [12]). However, it should be noted that although this ensures correct output, its purpose is to verify the results of an untrusted colouring algorithm by matching output instructions to corresponding input instructions and using live variable analysis to confirm that the results are correct. The algorithm itself is not formally verified, allowing for algorithms to be changed without needing to re-prove correctness (Tassarotti [13]).

A more rigorous approach to proving correctness is given in Blazy, Robillard & Appel [3], where a fully verified OCaml implementation of the common Iterated Register Coalescing algorithm (George [5]) is demonstrated. The algorithm works by repeatedly simplifying the graph by removing vertices of degree less than K, the number of available registers,[1] then combining vertices sharing a preference edge as long as their combined degree is less than K. This is a particularly strong example of a verified register allocator, and was ultimately implemented in CompCert. However, the proof appears fairly tightly coupled, making it difficult to modify to experiment with different heuristics or add features.

The approach used in this project was to separate the sub-stages of allocation as cleanly as possible and build up towards an algorithm with similar performance to algorithms used in practice, arranging the proofs so that further improvements can be made and verified without much extra work. Although the algorithm verified is not quite as powerful as the IRC algorithm verified in [3], it represents a full end-to-end system where individual components can be improved with relative ease.

[1] As they have degree less than K, these registers can trivially be coloured once the rest of the graph has been K-coloured.

Chapter 4

Live variable analysis and clash graph generation

Introduction

In order to achieve a complete end-to-end verification, it was decided that the verified allocator should take in some representation of three-address code (typical of this stage in the compilation process), and output three-address code with registers allocated. The overall proof would therefore show that the behaviour of the resulting code was exactly the same as that of the source in all situations. For this reason, a simplified representation of three-address code and an evaluation function were developed and an algorithm to generate clash graphs was verified. This chapter explores the representation used, the way 'colouring' registers was represented and the development and verification of the clash graph generator, along with the way a 'correct' register allocation was defined. The proofs showing that these definitions are correct and linking them to the project's overall goal will follow in Chapter 6.

4.1 Three-address code representation

Here we explore the simplified representation of three-address code, how evaluation was defined and how colourings were modelled and applied.

4.1.1 Registers and instructions

In real systems, register naming conventions tend to vary between processors, and there are often multiple groups of registers available. For the purposes of verification, this was simplified by using natural numbers to refer to registers, as in practice these numbers could be constrained to the number of available registers and then trivially mapped to real register names. Furthermore, there will generally be a great many different operations available on a given machine, but as the focus is on register allocation the only thing that matters here is the registers read and those assigned. Compilers commonly use three-address code as an intermediate representation, so a given instruction essentially consists of two source registers and one destination register. This led to the very simplified instruction representation given below.[1]

```
inst = Inst of num ⇒ num ⇒ num
```

For the main verification task, this was entirely satisfactory as it captures all the information needed. Later work extended this representation slightly: for instance, the addition of preferences meant it was necessary to include move instructions so that these could be used to compute preference graphs. Extensions to the three-address representation are discussed in Chapter 7.

Code blocks were then represented as simple lists of these instructions, to simplify verification. The extension to more complex code featuring basic blocks is not too great a leap, and is discussed in Section 8.2.1. The main difference in using this representation is that live variable analysis is somewhat simpler.

[1] This is a HOL datatype declaration in which the type name is given on the left followed by a series of constructors. Here there is only one constructor, `Inst`, which takes three values of type `num` representing the registers the instruction uses.

4.1.2 Code evaluation

As mentioned, it was not necessary for verification to have a wide range of instructions. The register allocation can be proven correct simply by showing that evaluating each instruction with an arbitrary binary operator and storing the result in the destination register has the same effect as it did before allocation. The precise meaning of 'has the same effect' is explained in Section 6.2, where it is proven to hold for the register allocator built during this project. The evaluation function used is given below, where f is an arbitrary binary operator and s is the store, a total function representing the values of all registers:

```
eval f s [] = s
eval f s (Inst w r1 r2::code) =
eval f ((w =+ f (s r1) (s r2)) s) code
```

Each instruction is evaluated by looking up the two source registers in the store and applying the binary operation to them, then updating the store function to map the destination register to the resulting value. The base case, where we have reached the end of the code block, simply returns the store as it is.

4.1.3 Colourings

With the basic code type defined, it was then necessary to model a register allocation abstractly in order to reason about allocations conveniently. An allocation was defined as a 'colouring', in keeping with the focus on graph-colouring approaches, mapping registers to registers. It thus made sense simply to use a total function of type $num \rightarrow num$. This made it very easy to later define properties of colourings – most importantly, the notion of a colouring not mapping conflicting virtual registers to the same target register. This definition is discussed in Section 4.2.3.

Applying a colouring is then very simple: the function iterates through instructions, applying the colouring function to each register as it goes. The

definition is given below:

```
apply c [] = []
apply c (Inst w r1 r2::code) =
Inst (c w) (c r1) (c r2)::apply c code
```

For example, a sample colouring function for the graph in Chapter 2 is given below:

$1 \rightarrow 1$

$2 \rightarrow 3$

$3 \rightarrow 2$

$4 \rightarrow 0$

$5 \rightarrow 2$

$6 \rightarrow 0$

Thus the graph resulting from applying this colouring is as follows:

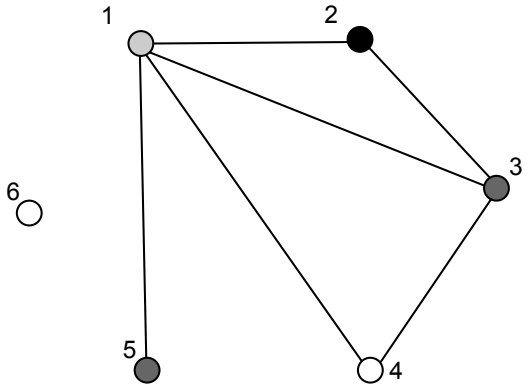

Figure 4.1: Clash graph generated for the sample code, after colouring

The resulting code, which now uses only four registers, is given below:

```
R1 = R3 + R2
R0 = R1 * R3
R2 = R2 - R0
R0 = R1 + R2
```

This completes the definition of the basic primitives used to represent code in the proofs.

4.2 Live variables and clash graphs

This section describes the definitions and functions used in producing a verified clash graph generator, and covers the lemmas and proofs used to prove its correctness. The first subsection demonstrates the representation of sets using lists, which was adopted to simplify definitions and proofs, and many of the properties proved of such sets which were vital in later proofs. Verifying a clash graph generator was then handled in two stages: first, a live variable analysis was implemented and proven correct. A clash graph generator was then developed using the live variable analysis, and various important properties were verified to aid later proofs. Section 4.2.3 details how the correctness of a generated colouring was defined, a definition which was fundamental to the rest of the project.

4.2.1 Set representation and proofs

As mentioned, it was decided that for the purposes of the project it would be most convenient to represent sets as lists rather than using HOL's set primitives. Using lists made it considerably easier to define functions iterating over elements of a set, and made many of the proofs and definitions a lot easier to work with. Furthermore, lists allow for an ordering to exist, which was particularly useful when applying heuristics to vertices of clash graphs. The downside was that a number of properties of sets had to be verified, and showing that various set operations preserved its duplicate-freeness took a lot of extra effort. (The necessary proofs and definitions related to duplicate-freeness are given in Section 4.2.3.)

The two basic operations on sets are insertion and deletion of elements. The definitions of these for 'list sets' are given below:

```
insert x xs = if MEM x xs then xs else x::xs
delete x xs = FILTER (λ y. x ≠ y) xs
```

Insertion simply tests for membership and places the new element at the head of the list if it is not a member, and deletion filters the list by inequality to the element to be removed. Several properties were then proven to demonstrate that these definitions work:

```
⊢ MEM x (insert x list)
```

This just states that inserting an element does indeed add it – it is correct with respect to `MEM`. The proof is a simple case split on whether the value being added is already a member of the set.

```
⊢ MEM x (insert y list) ∧ x ≠ y ⇒ MEM x list
```

This states that if x is a member of a set with a value inserted, and it is not equal to that value, then it must be a member of the original set. The proof is a trivial case split on whether y belongs to the original set, simplifying definitions in each case to prove the goal.

$$\frac{\texttt{MEM } x \ xs}{\texttt{MEM } x \ \texttt{(insert } y \ xs\texttt{)}}$$

The proof of this is a simple expansion of the definition of insertion, and shows that inserting an element retains the elements already there.

$$\frac{a \notin \texttt{set (insert } x \ xs\texttt{)}}{a \notin \texttt{set } xs}$$

The contrapositive of the above, this is a useful way of simplifying goals where some value isn't an element of a generated set.

```
⊢ x ∉ set (delete x list)
```

The basic proof of delete's correctness, this shows that deleting a value from a set does in fact remove it. The proof follows by expanding the definitions of `delete` and `filter`.

It was also necessary to implement set union, and union over multiple sets. The definition of set union is given below:

```
list_union [] ys = ys
list_union (x::xs) ys = insert x (list_union xs ys)
```

It was necessary to show that this definitely contains all elements from both sets. The relevant property is given below, and was proven by induction on the size of the first set in the union by demonstrating that once x is reached it is guaranteed to be included in the result.

$$\frac{\texttt{MEM}\ x\ \mathit{list} \vee \texttt{MEM}\ x\ \mathit{list}'}{\texttt{MEM}\ x\ (\texttt{list\_union}\ \mathit{list}\ \mathit{list}')}$$

Union over multiple sets, here referred to as a flattening union, is defined by recursively applying `list_union` to pairs of sets from the end of the list forwards:

```
list_union_flatten [] = []
list_union_flatten (l::ls) =
list_union l (list_union_flatten ls)
```

Again, a completeness proof was required for later proofs. This is given below:

$$\vdash \texttt{MEM}\ \mathit{list}\ \mathit{lists} \wedge \texttt{MEM}\ x\ \mathit{list} \Rightarrow \texttt{MEM}\ x\ (\texttt{list\_union\_flatten}\ \mathit{lists})$$

The proof used the same technique as the last one, combined with the fact that the union of two lists contains all elements from both lists (as was just proved).

This completes the basic definitions and proofs for list-based sets. An intersection operation was not required for the sets used, as there was no function in the algorithm which used it. Additionally, only completeness of unions was required – interestingly, it was not necessary to prove that they didn't introduce any extra elements for any later proofs. This is a side-effect of all live variable analysis methods generally computing safe over-approximations:

an algorithm which satisfies an overly large set of conflicts will automatically satisfy a smaller, more precise set.

4.2.2 Live variable analysis

As mentioned earlier, the decision to represent a block of code as a simple list of instructions makes live variable analysis noticeably less complex. The definition used is given below:

```
get_live [] live = live
get_live (Inst w r1 r2::code) live =
insert r1 (insert r2 (delete w (get_live code live)))
```

This is an implementation of the liveness equation given in Section 2, which also allows for some variables to be live at the end of the program (which can be used to link together analyses for adjacent blocks of code, or specify which registers hold program results). Showing that this definition is correct is subtle: correctness with respect to register allocation is shown later by demonstrating that colourings satisfying constraints built up from this definition will not change code behaviour. However, another way of showing correctness is by proving that only the variables returned by `get_live` will affect the evaluation of a program, a property which is stated below:

$$\frac{\texttt{MAP } s \texttt{ (get\_live } code\ live\texttt{) = MAP } t \texttt{ (get\_live } code\ live\texttt{)}}{\texttt{MAP (eval } f\ s\ code\texttt{) } live \texttt{ =MAP (eval } f\ t\ code\texttt{) } live}$$

The antecedent states that the two stores s and t agree on the variables which are live at the start of the code. The consequent then claims that the stores at the end of the program's evaluation will agree on whatever variables are designated as live at the end. The proof of this statement is by induction on the size of the code, expanding all terms and rewriting using the lemmas about insertion and deletion proved earlier along with many of the standard HOL list axioms.

4.2.3 Defining correctness of a colouring

In order to be able to prove that generated colourings are correct, and that correct colouring do not affect code behaviour, defining the concept of a 'correct' colouring was essential. Fundamentally, a valid colouring is one which will not map two simultaneously live variables to the same target register. That is to say, for the set of variables live at any given instruction, mapping the colouring over them will not result in any duplicates. To facilitate this sort of definition, the property of being duplicate-free was defined as a function and a number of properties were proven of this. These are outlined in the next subsection.

Duplicate-freeness and associated lemmas

Duplicate-freeness of a list was defined using the following function:

$$\texttt{duplicate\_free []} \iff \texttt{T}$$
$$\texttt{duplicate\_free } (x :: xs) \iff x \notin \texttt{set } xs \wedge \texttt{duplicate\_free } xs$$

To open up more options for proving duplicate-freeness, a pair of lemmas describing its behaviour in terms of element equality were also proved. The first shows that if no two different elements of the list are equal, it is duplicate-free:

$$\begin{array}{l}\vdash (\forall x\ y.\\ \quad x < \texttt{LENGTH } list \wedge y < \texttt{LENGTH } list \wedge x \neq y \Rightarrow\\ \quad \texttt{EL } x\ list \neq \texttt{EL } y\ list) \Rightarrow\\ \texttt{duplicate\_free } list\end{array}$$

The second comes from the other side, stating that if a list is duplicate-free, no two elements will be equal:

$$\begin{array}{l}\vdash \texttt{duplicate\_free } list \wedge x < \texttt{LENGTH } list \wedge y < \texttt{LENGTH } list \wedge\\ x \neq y \Rightarrow\\ \texttt{EL } x\ list \neq \texttt{EL } y\ list\end{array}$$

The proof was fairly long as there were several details to account for, but essentially boiled down to an induction where the inductive step case splits on x or y being equal to zero. If neither is, the inductive hypothesis proves the goal; otherwise, the other must be greater than zero and the definition of duplicate-freeness shows that the value at zero cannot be a member of the rest of the list.

It was then necessary to prove that the set operations defined on list sets preserve duplicate-freeness, as required. The statements for insertion and deletion follow:

$\vdash$ `duplicate_free (insert` n $list$`)` $\iff$ `duplicate_free` $list$

$\vdash$ `duplicate_free` $list$ $\Rightarrow$ `duplicate_free (delete` n $list$`)`

The first follows from the definitions, and the second was shown by induction.

Duplicate-freeness was then proven for unions of duplicate-free sets, first by proving that it holds for a simple union of two list sets:

$\vdash$ `duplicate_free` xs $\wedge$ `duplicate_free` ys $\Rightarrow$
`duplicate_free (list_union` xs ys`)`

This was shown by induction, using the fact that unions rely on insertion which preserves duplicate-freeness.

This was then extended to general unions. The proof works by induction, using the above lemma and expanding definitions.

$$\frac{\texttt{EVERY } (\lambda\, list.\ \texttt{duplicate\_free } list)\ lists}{\texttt{duplicate\_free (list\_union\_flatten } lists\texttt{)}}$$

The next objective was to show that the set of registers returned by `get_live` contains no duplicates, assuming there are no duplicates in the set of registers considered to be live at the end of the code. This was a fairly simple proof by induction, using the lemmas about duplicate-freeness of insertion and deletion and the definition of `get_live`:

$$\frac{\texttt{duplicate\_free}\ \textit{live}}{\texttt{duplicate\_free (get\_live}\ \textit{code}\ \textit{live}\texttt{)}}$$

Finally, the following useful property was proved, stating that if the result of mapping a function over a list is duplicate-free and two elements of the original list are not equal then the function maps them to different things:

$$\begin{array}{l}\vdash \texttt{duplicate\_free (MAP}\ c\ \textit{live}\texttt{)} \wedge x \neq y \wedge \texttt{MEM}\ x\ \textit{live} \wedge \\ \quad \texttt{MEM}\ y\ \textit{live} \Rightarrow \\ \quad c\ x \neq c\ y\end{array}$$

This lemma was particularly useful in proving injectivity properties of valid colourings later. The proof is by induction on the list *live*, where in the inductive case we expand the definitions and case split on whether x or y is equal to the current head of the list. In either case, the other must be behind in the list and inequality of cx and cy follows by the definition of duplicate-freeness.

Basic definition of colouring correctness

As discussed earlier, a colouring is deemed 'correct' if for any instruction's set of live variables, the result of mapping the colouring function over it is duplicate-free. This is relatively easy to define:

$$\begin{array}{l}\texttt{colouring\_ok}\ c\ \texttt{[]}\ \textit{live} \iff \texttt{duplicate\_free (MAP}\ c\ \textit{live}\texttt{)} \\ \texttt{colouring\_ok}\ c\ \texttt{(Inst}\ w\ r_1\ r_2\texttt{::}\textit{code}\texttt{)}\ \textit{live} \iff \\ \texttt{duplicate\_free (MAP}\ c\ \texttt{(get\_live (Inst}\ w\ r_1\ r_2\texttt{::}\textit{code}\texttt{)}\ \textit{live}\texttt{))} \wedge \\ \texttt{colouring\_ok}\ c\ \textit{code}\ \textit{live}\end{array}$$

An important property of colouring correctness is that it is also correct for a subset of the code block, and this statement is given below. The proof of this is a trivial expansion of the definition of `colouring_ok`.

$$\begin{array}{l}\vdash \texttt{colouring\_ok}\ c\ \texttt{(Inst}\ n\ n_0\ n_1\texttt{::}\textit{code}\texttt{)}\ \textit{live} \Rightarrow \\ \quad \texttt{colouring\_ok}\ c\ \textit{code}\ \textit{live}\end{array}$$

This definition works nicely for the proofs working directly with three-address code, and was used for most of the later proofs regarding code evaluation behaviour. However, it is very strongly tied to the code definitions and doesn't clearly reflect the conflicts graph colouring is working with, making proofs relating graph colourings to valid register allocations particularly difficult.

For this reason, an alternative definition of `colouring_ok` was created and proved to be equivalent to the original definition. This alternative definition was much easier to work with when reasoning about clash graphs, as demonstrated later in Section 6.1.2.

Alternative `colouring_ok` definition

The alternative colouring_ok definition was designed to be slightly more abstract than the original one: rather than directly working with `get_live`, it uses a function `colouring_respects_conflicting_sets` to determine whether a colouring respects an arbitrary list of conflicting sets. Here we say a colouring respects a conflicting set if mapping it over the elements of the set produces a duplicate-free result. The definition of this function is given below:

```
colouring_respects_conflicting_sets c [] ⟺ T
colouring_respects_conflicting_sets c (s::ss) ⟺
duplicate_free (MAP c s) ∧
colouring_respects_conflicting_sets c ss
```

This can also be conveniently expressed using HOL's `EVERY` function, as shown in the following theorem:

```
⊢ colouring_respects_conflicting_sets c sets ⟺
  EVERY (λlist. duplicate_free (MAP c list)) sets
```

The proof is a trivial induction on the number of conflicting sets where each case is solved by expanding definitions and using `METIS_TAC`.

Clearly the conflicting sets to use in this case are just the sets of live variables at each instruction. To keep things abstract, this was pulled into a separate

function, defined as shown below:

```
conflicting_sets [] live = [live]
conflicting_sets (Inst w r1 r2::code) live =
get_live (Inst w r1 r2::code) live::conflicting_sets code live
```

With this in hand, `colouring_ok_alt` can be defined by passing the results of this function into `colouring_respects_conflicting_sets`:

```
colouring_ok_alt c code live ⟺
colouring_respects_conflicting_sets c
  (conflicting_sets code live)
```

The advantage of abstracting details in this way is that we can prove properties of `conflicting_sets` and `colouring_respects_conflicting_sets` which allow us to avoid working directly with instructions altogether. This makes it easier to make changes to the way code is represented, as is done for some of the extensions in Chapter 7.

For these two definitions to be used interchangeably in proofs, it was necessary to prove that they were equivalent:

```
⊢ colouring_ok_alt c code live ⟺ colouring_ok c code live
```

The proof is a trivial induction where the inductive case is shown by simplification.

4.2.4 Clash graph generation

With the verified live variable analysis complete, the next step was to use this to generate clash graphs which could be fed to the graph colouring algorithms, and to verify that these graphs accurately reflected the register conflicts present in the code.

Initial work here focussed on finding the graph representation which would make the definitions and proofs easiest to work with. A number of alternatives were considered, but it was ultimately decided that the best option

would be a set of (v, es) pairs where v is a vertex and es is the set of vertices with which v shares an edge. This works best because it allows vertices to be considered in any order and lets the code quickly look up which vertices it conflicts with without having to traverse the graph. Representing conflicts between vertices using a HOL relation (that is, a function of type $num \times num \to bool$) was considered, but this made it difficult to iterate over conflicting registers and was more awkward to construct given the results of live variable analysis.

Computing conflicts for a register

The first step in generating such a graph was to generate a list set of all the conflicts for a particular register. This used the following function:

```
conflicts_for_register r code live =
delete r
  (list_union_flatten
     (FILTER (λset. MEM r set) (conflicting_sets code live)))
```

The definition is slightly complex. The function first finds the list of conflicting sets for the code using the `conflicting_sets` function defined in the previous subsection. It then filters these down to only those sets which the current register is a member of, and applies `list_union_flatten` to combine these into a single set of all registers belonging to a conflicting set which the current register also belongs to. Finally, it removes the current register, as a register should not appear in its own list of conflicts, and returns the result.

The result is a duplicate-free set list containing all the registers a given register conflicts with. Its duplicate-freeness is verified below, and was essential in later proofs:

```
⊢ duplicate_free live ⇒
  duplicate_free (conflicts_for_register r code live)
```

To prove this we need the following property of conflicting sets, which is a simple induction using the fact that `get_live` is duplicate-free:

```
⊢ duplicate_free live ⇒
  EVERY (λ list. duplicate_free list)
    (conflicting_sets code live)
```

We can then show that filtering this set yields a duplicate-free set using the following property of `EVERY`, verified by trivial induction:

```
⊢ EVERY P list ⇒ EVERY P (FILTER Q list)
```

The goal then follows as the flattening union function and deletion both preserve duplicate-freeness.

Constructing the graph

Once the function to determine a register's conflicts had been defined, the function to construct the whole graph was fairly simple to construct:

```
get_conflicts code live =
MAP (λ reg. (reg,conflicts_for_register reg code live))
  (get_registers code live)
```

This just maps the `conflicts_for_register` function over all registers used in the program, putting the results into a tuple with the number of the register. The result is a graph of the following form:

$[(r_1, [c_1, \ldots, c_n]), \ldots, (r_n, [c_1, \ldots, c_n])]$

Here r_n is the n^{th} register and c_n is the n^{th} conflicting register. The definition depends on a supporting function which finds the set of all registers used by a block of code:

```
get_registers [] live = live
get_registers (Inst r0 r1 r2::code) live =
insert r0 (insert r1 (insert r2 (get_registers code live)))
```

Note that this is trivially duplicate-free where *live* is duplicate-free, because it only uses set insertions and these have been proven to preserve duplicate-freeness:

$\vdash$ `duplicate_free` *live* $\Rightarrow$
`duplicate_free (get_registers` *code live*`)`

Proof of correctness

This section contains the necessary proofs demonstrating that the clash graph generator works as it should. Three properties were shown: registers not included in the results of `get_registers` do not feature in any program instruction, in `get_live` or in any conflicting set (and so no vertices are missing from the graph), registers do not conflict with themselves, and any pair of registers in the same conflicting set will appear in each other's list of conflicts. These theorems are revisited later in Chapter 6, where they are used to help connect the graph colouring proofs to the proofs about code and prove the overall goal of the project.

The first thing to prove was that registers not included in `get_registers` are never used in any program instruction:

$\vdash$ $r \notin$ `set (get_registers` *code live*`)` $\wedge$
`MEM (Inst` w r_1 r_2`)` *code* $\Rightarrow$
$r \neq w \wedge r \neq r_1 \wedge r \neq r_2$

The proof of this statement is by induction and expanding definitions.

It follows that an unused register will never feature in the results of `get_live`, as `get_live` only inserts registers which are used in instructions:

$\vdash$ $r \notin$ `set (get_registers` *code live*`)` $\Rightarrow$
$r \notin$ `set (get_live` *code live*`)`

Using the above two theorems, it was possible to prove that an unused register does not feature in any conflicting set by induction on the code:

$\vdash$ $r \notin$ `set (get_registers` *code live*`)` $\wedge$
`MEM` *set* `(conflicting_sets` *code live*`)` $\Rightarrow$
$r \notin$ `set` *set*

A consequence of the last theorem is that evaluating `get_conflicts` on an

unused register will yield the empty list, a property which allows us to specify definitions over all registers without caring about those which don't get used in the code. This useful property is given below:

$\vdash$ $r \notin$ `set (get_registers` *code live*`) ⇒`
`(conflicts_for_register` r *code live* `= [])`

Having effectively shown that `get_registers` covers all the registers that need to be handled, the next stage in verifying the clash graph generator was to show that a register never appears in its own list of conflicts. This proof follows easily from the definition of `conflicts_for_register` as the function removes the register at the end:

$\vdash$ $r \notin$ `set (conflicts_for_register` r *code live*`)`

The proof uses the following simple lemma, proved by induction on the list:

$\vdash$ $x \notin$ `set (FILTER (`$\lambda\, y$`.` $x \neq y$`)` *list*`)`

To complete the correctness proof, we now verify that generated clash graphs are complete in that any two registers in the same conflicting set will appear in each other's list of conflicts:

$\vdash$ `MEM` c `(conflicting_sets` *code live*`)` $\wedge$ `MEM` r c $\wedge$ `MEM` s c $\wedge$
$r \neq s$ ⇒
`MEM` r `(conflicts_for_register` s *code live*`)`

The proof depends on the following lemma, which states that if a list of lists is being filtered for whether they contain x, and x is in *list*, *list* is in the result:

$\vdash$ `MEM` *list lists* $\wedge$ `MEM` x *list* ⇒
`MEM` *list* `(FILTER (`λ *list*`.` `MEM` x *list*`)` *lists*`)`

This can be shown by induction, case splitting on whether the current list head is equal to the list containing x. The goal above now follows by using this lemma to show that both registers appear in the result of filtering conflicting sets by membership, and by simplifying definitions they can be shown to appear in the overall result of `conflicts_for_register`.

This concludes the proof of correctness of the clash graph generator. These theorems will be used later to help prove the overall correctness of the register allocator in Chapter 6.

Chapter 5

Colouring algorithms

Introduction

The core verification task of this project is, of course, to prove the correctness of the algorithms that perform the register allocation. In this chapter the focus is on the algorithms and heuristics used to colour clash graphs, how correctness was modelled and how it was verified for the algorithms used. In the interests of having a minimal algorithm verified early on, a simple colouring algorithm which simply assigned each vertex its own colour was verified first. The next objective was to verify an algorithm which re-uses old colours wherever possible by assigning to each vertex the lowest register which doesn't conflict with its neighbours. From there, the work focussed on extra heuristics which could improve the performance of the lowest-first greedy algorithm by considering vertices in a different order.

5.1 Modelling colouring correctness

5.1.1 Defining correctness

Much as correctness of colouring was defined in terms of code with `colouring_ok` and `colouring_ok_alt`, correctness with respect to a clash graph was defined through the function `colouring_satisfactory`[1]:

`colouring_satisfactory` *col* `[]` $\iff$ `T`
`colouring_satisfactory` *col* `((`*r*`,`*rs*`)::`*cs*`)` $\iff$
col *r* $\notin$ `set (MAP` *col* *rs*`)` $\wedge$ `colouring_satisfactory` *col* *cs*

This simply states that colouring a vertex and all its neighbours will not lead to it having the same colour as any of its neighbours, as required for a correct colouring. It thus breaks the proof that a given colouring is correct into two steps: it needs to respect the set of conflicts for the current vertex, and it needs to work for the rest of the vertices of the graph. As the colouring algorithms colour one vertex at a time, the second part of the proof should intuitively be correct as the current set of conflicts represent all the constraints on the current vertex. However, it was fairly difficult to verify in practice, as discussed in some of the later sections in this chapter.

5.1.2 Requirements on clash graphs

For colouring algorithms to work properly, the clash graphs need to be *well-formed* in that every edge list is well-formed according to the following definition:

`edge_list_well_formed (`*v*`,`*edges*`)` $\iff$
v $\notin$ `set` *edges* $\wedge$ `duplicate_free` *edges*

This specifies that a vertex cannot be linked to itself, and the set of vertices it is adjacent to cannot have any duplicates. Overall well-formedness is then given by the following definition:

[1] The proof that this is equivalent follows in Chapter 6

```
graph_edge_lists_well_formed es ⟺
EVERY (λx. edge_list_well_formed x) es
```

Another important property is that no vertex appears twice in the graph. This property is stated below:

```
graph_duplicate_free [] ⟺ T
graph_duplicate_free ((r,rs)::cs) ⟺
(∀rs'. (r,rs') ∉ set cs) ∧ graph_duplicate_free cs
```

Finally, each edge list has to contain *all* vertices to which the current one is joined – in other words, if a vertex r_1 appears in the edge list for r_2 (rs_2), then r_2 must also be present in the edge list for r_1 (rs_1):

```
graph_reflects_conflicts cs ⟺
∀r1 r2 rs1 rs2.
  MEM (r1,rs1) cs ∧ MEM (r2,rs2) cs ∧ MEM r1 rs2 ⇒ MEM r2 rs1
```

In Section 6.1.1, the proof that graphs generated by `get_conflicts` have these required properties is described, but for now it is assumed that they will hold of any graph passed in to the colouring algorithms.

5.2 Naïve graph colouring

A very naïve graph colouring algorithm was verified first to obtain a complete proof as early into the project as possible which could be built upon later. This section describes the definition and verification of this simple colouring approach.

5.2.1 The algorithm

The naïve algorithm simply allocates a colour for every register and assigns them in ascending order. This uses an auxiliary function, `naive_colouring_aux`:

```
naive_colouring_aux [] n = (λx. n)
naive_colouring_aux ((r,rs)::cs) n =
(r =+ n) (naive_colouring_aux cs (n + 1))
```

This algorithm assigns registers from n upwards. The main naïve colouring function simply calls this starting from zero:

```
naive_colouring constraints = naive_colouring_aux constraints 0
```

The result is a very simple colouring method which is intuitively correct. However, verifying its correctness was slightly harder than expected, relying on several lemmas about `naive_colouring_aux`.

5.2.2 Correctness proof

Correctness of the algorithm was proved through correctness of `naive_colouring_aux`. The two statements of correctness follow. Note that only `graph_edge_lists_well_formed` was required of input graphs here, as the algorithm is very simple.

```
⊢ graph_edge_lists_well_formed cs ⇒
  ∀n. colouring_satisfactory (naive_colouring_aux cs n) cs
⊢ graph_edge_lists_well_formed cs ⇒
  colouring_satisfactory (naive_colouring cs) cs
```

The second clearly follows from the first by expanding definitions and instantiating variables:

```
⊢ (∀n. colouring_satisfactory (naive_colouring_aux cs n) cs) ⇒
  colouring_satisfactory (naive_colouring cs) cs
```

Thus we need only prove correctness of `naive_colouring_aux`. This required several lemmas describing the behaviour of the function and related definitions.

The first main goal was to show that each colour assigned by the algorithm has not been used before:

```
⊢ naive_colouring_aux cs (n + 1) x ≠ n
```

To prove this, consider first the following lemma equating two evaluations of `naive_colouring_aux` starting from separate values:

```
⊢ (naive_colouring_aux cs (n + 1) x =
    naive_colouring_aux cs n x) ∨
   (naive_colouring_aux cs (n + 1) x =
    naive_colouring_aux cs n x + 1)
```

This is a clear consequence of the definition: assigned colours starting from $n + 1$ are either equal to or one greater than those found by starting from n. They are equal in the case where the register is unused (and so is simply mapped to itself by the base case of the definition), and one greater in all other cases. This was shown by induction on the graph. From this we can prove that all assigned colours are greater than the starting value by induction, using the last lemma to link the result of the inductive hypothesis to the goal:

```
⊢ naive_colouring_aux cs (n + 1) x > n
```

The original lemma is a simple consequence of this:

```
⊢ naive_colouring_aux cs (n + 1) x ≠ n
```

The second main lemma specifies that updating the mapped value of a function f for a value not featured in *list* means mapping the updated f over *list* is the same as mapping f over *list*:

```
⊢ x ∉ set list ⇒ (MAP ((x =+ n) f) list = MAP f list)
```

This was proved by inducting on the list and showing that the function's update couldn't affect any element. From this, a useful lemma for updating `naive_colouring_aux` was deduced, stating that colouring an unused vertex will not affect the result of applying the colouring:

```
⊢ q ∉ set r ⇒
   (MAP ((q =+ n) (naive_colouring_aux cs (n + 1))) r =
    MAP (naive_colouring_aux cs (n + 1)) r)
```

The third lemma to be proved is as follows:

$\vdash$ (∀x. f x ≠ n) ⇒ n ∉ set (MAP f *list*)

This just states that if a function is never equal to n, n will not feature in the result of mapping f over a list. The proof is a simple induction using the definitions of MEM and MAP.

The final lemma claims that a satisfactory naïve colouring starting from $n+1$ will still be satisfactory after mapping a register to n, as n will not have been used by any other register:

```
⊢ graph_edge_lists_well_formed cs ∧
  colouring_satisfactory (naive_colouring_aux cs (n + 1)) cs ⇒
  colouring_satisfactory
    ((q =+ n) (naive_colouring_aux cs (n + 1))) cs
```

The proof is fairly simple: n is never used, as proved earlier, and intuitively using a colour which has never been used will not cause any clashes. Thus the update is guaranteed to be safe. However, the intuitive statement here was more difficult to prove. A full statement of it is given in the following lemma:

```
⊢ graph_edge_lists_well_formed cs ∧
  colouring_satisfactory c cs ∧ (∀x. c x ≠ n) ⇒
  colouring_satisfactory ((q =+ n) c) cs
```

The proof depends on three sub-lemmas:

Outputs cannot exist without being mapped to

```
⊢ f x ∉ set (MAP f list) ∧ f x ≠ n ⇒
  f x ∉ set (MAP ((q =+ n) f) list)
```

This statement is slightly complex, but essentially says that if a value is not in the result of mapping a function over a list and it is not equal to n, then it will still not be in the result if we update the function to map something to n. The proof depends on the following lemma:

$\vdash$ (q =+ n) f h $\neq$ f h $\Rightarrow$ (h = q)

This can be shown by case splitting on $h = q$ and simplifying. The proof of the original statement now follows using this lemma by induction on the list. We show show that if the head of the list, h, contradicts the goal then we must have $(q = +n)fh = fx$ and thus not equal to n, and also $fh \neq fx$. However, this means $(q = +n)fh \neq fh$ and thus $h = q$ by the lemma. Therefore $fh = n$, contradicting the above.

Map output only contains values mapped from inputs

$\vdash x \notin$ set *list* $\wedge$ ($\forall y.\ y \neq x \Rightarrow f\ y \neq n$) $\Rightarrow n \notin$ set (MAP *f list*)

Simply put, if only one value x maps to a particular output value n, mapping the function over a list which does not contain x will give a list not containing n. The proof is by induction, and is trivial using the induction hypothesis and expanding the definitions of MEM and MAP.

Applying an update only changes the updated value

$\vdash$ ($\forall x.\ f\ x \neq n$) $\Rightarrow \forall x.\ x \neq w \Rightarrow$ (w =+ n) f x $\neq$ n

That is, if a function is never equal to n, updating it so one value is mapped to n does not affect any other inputs. The proof is trivial by case splitting on $x = w$ in the conclusion.

These three sub-lemmas are sufficient to prove the lemma on page 34. We induct on the graph, and do a case split on whether the current vertex is equal to the one being updated. If it is, we use the fact that the new colour will not have been used before and the current vertex is not in its conflicts list to show that it is satisfactory on the current vertex, and the inductive hypothesis to prove the remainder of the goal. If it is not, we use the fact that the current vertex does not belong to its list of conflicts to show that its colour cannot conflict with anything in the set, and we again use the inductive hypothesis to prove the rest.

It is now possible to prove the original statement of `naive_colouring_aux` being satisfactory:

$$\vdash \texttt{graph\_edge\_lists\_well\_formed}\ cs \Rightarrow \\ \forall n.\ \texttt{colouring\_satisfactory (naive\_colouring\_aux}\ cs\ n\texttt{)}\ cs$$

As mentioned when defining `colouring_satisfactory` in Section 5.1.1, proving a colouring satisfactory consists of two goals: showing that an update doesn't violate the constraints on its vertex, and showing that if the previous colouring was valid for the rest of the graph then the updated one will be too. This latter goal follows from the lemma proved earlier:

$$\vdash \texttt{graph\_edge\_lists\_well\_formed}\ cs \wedge \\ \texttt{colouring\_satisfactory}\ c\ cs \wedge (\forall x.\ c\ x \neq n) \Rightarrow \\ \texttt{colouring\_satisfactory ((}q\ \texttt{=+}\ n\texttt{)}\ c\texttt{)}\ cs$$

The assumption that the original colouring is never equal to n follows from the lemma proved earlier:

$$\vdash \texttt{naive\_colouring\_aux}\ cs\ \texttt{(}n\ \texttt{+ 1)}\ x \neq n$$

The first goal was then proved using the assumption that graph edge lists are well-formed and a combination of the other lemmas proved here.

Thus the verification of `naive_colouring_aux` is complete, and by the implication shown at the start of this section we have that the overall algorithm is correct:

$$\vdash \texttt{graph\_edge\_lists\_well\_formed}\ cs \Rightarrow \\ \texttt{colouring\_satisfactory (naive\_colouring}\ cs\texttt{)}\ cs$$

5.2.3 Example

To demonstrate that this algorithm works as expected, the function resulting from running it on the sample code given in Chapter 2 is given below, along with the resulting code after colouring:

$1 \to 4$
$2 \to 0$
$3 \to 1$
$4 \to 2$
$5 \to 5$
$6 \to 3$

```
R4 = R0 + R1
R2 = R4 * R0
R5 = R1 - R2
R3 = R4 + R5
```

This is using exactly the same number of registers as the source code, as a new register is assigned to every virtual register in the order in which they are considered.

5.3 Lowest-first colouring

The lowest-first algorithm is an improvement over naïve colouring which allows colours to be re-used where possible. Essentially, the function examines the vertices of the graph in an arbitrary order and assigns to each the lowest possible colour which does not conflict with the colours already assigned to any of its neighbours. This section explores the definition of this function and the proof of correctness, again using `colouring_satisfactory` as the verification condition.

5.3.1 The algorithm

The main algorithm is fairly simple: to colour a particular vertex, colour all those after it in the list and then find the lowest colour which doesn't conflict with the colours that have been assigned to its neighbours. The definition is given here:

```
lowest_first_colouring [] = (λx. 0)
lowest_first_colouring ((r,rs)::cs) =
(let col = lowest_first_colouring cs in
 let lowest_available = lowest_available_colour col rs
 in
   (r =+ lowest_available) col)
```

The function `lowest_available_colour` is fairly self-explanatory:

```
lowest_available_colour col cs =
smallest_nonmember 0 (MAP col cs)
```

This uses the auxiliary function `smallest_nonmember`, which begins from the starting value x and increments until it is not a member of the supplied list, returning the first non-member. The definition is supplied below:

```
smallest_nonmember x list =
if MEM x list then smallest_nonmember (x + 1) list else x
```

Termination of `smallest_nonmember`

The `smallest_nonmember` function's termination is non-trivial, and so had to proved separately. The approach used was to show that each recursive iteration decreases the distance between the smallest nonmember value and the maximum value of the list, found using the following function:

```
list_max [] = 0
list_max (x::xs) =
(let tail = list_max xs in if x > tail then x else tail)
```

To make the proof easier, the following lemma showing that the maximum of a list is indeed its maximum was proved by induction:

$\vdash$ `MEM` x *list* $\Rightarrow$ $x \leq$ `list_max` *list*

It is fairly simple to show that as the value to be returned is incremented on each recursion, it will continuously move towards the maximum of the

list, and so termination of `smallest_nonmember` can now be proved using a relation which subtracts the value of x from the maximum of the list.

5.3.2 Correctness proof

As for the naïve colouring, the proof of correctness depends on showing two things: the colour selected for a new vertex will not conflict with any of its neighbours, and the updated colouring will be valid for all vertices whose colours have already been assigned. The first of these amounts to proving that the value returned by `smallest_nonmember` is indeed not a member:

$\vdash$ `smallest_nonmember` *n list* $\notin$ `set` *list*

The proof is trivial, using the following custom induction rule generated for `smallest_nonmember` and case-splitting on whether x is a member of the list:

$\vdash$ (∀ *x list*. (`MEM` *x list* ⇒ *P* (*x* `+ 1`) *list*) ⇒ *P x list*) ⇒
∀ *v* v_1. *P v* v_1

It is now possible to show that the value selected by `lowest_available_colour` is valid given the set of constraints:

$\vdash$ `lowest_available_colour` *col cs* $\notin$ `set (MAP` *col cs*`)`

The proof follows by expanding the definition of `lowest_available_colour` and using the previous lemma.

The following lemma was also useful in the proof. It claims that if a colouring update is valid for the current vertex's set of conflicts, and the colouring was valid for all other vertices before updating, then the updated colouring will be valid for all other vertices:

$\vdash$ `graph_reflects_conflicts ((`*r*`,`*rs*`)::`*cs*`)` ∧
`graph_duplicate_free ((`*r*`,`*rs*`)::`*cs*`)` ∧
n $\notin$ `set (MAP ((`*r* `=+` *n*`)` *col*`)` *rs*`)` ∧
`colouring_satisfactory` *col cs* ⇒
`colouring_satisfactory ((`*r* `=+` *n*`)` *col*`) ((`*r*`,`*rs*`)::`*cs*`)`

This follows intuitively from the fact that no vertex can appear twice in a graph and the conflicts listed for the current vertex cover all conflicts it has with other vertices. The proof uses one extra lemma expressing that `colouring_satisfactory` means every vertex in the graph has no clashes:

```
⊢ colouring_satisfactory col cs ⟺
  EVERY (λ(x,xs). col x ∉ set (MAP col xs)) cs
```

This is used to simplify the goal and assumptions, and then the definitions of `graph_reflects_conflicts` and `graph_duplicate_free` are used to verify the goal. The key trick used is that for any vertex v and edge list es where the updated vertex is not present, no new conflicts are introduced. If the updated vertex is in the edge list es then v must have been in the edge list for the updated vertex by the definition of `graph_reflects_conflicts`, and therefore doesn't conflict by the assumption that the updated colouring respects the current vertex's set of conflicts. This lemma covers the second part of the proof, and makes it possible to complete the proof of the algorithm's correctness:

```
⊢ graph_reflects_conflicts cs ∧ graph_duplicate_free cs ∧
  graph_edge_lists_well_formed cs ⇒
  colouring_satisfactory (lowest_first_colouring cs) cs
```

The lowest-first colouring algorithm is therefore correct with respect to `colouring_satisfactory`. This is a reasonably good algorithm, but it is still fairly simplistic. In the next section, various heuristics which can further improve performance will be explored.

5.3.3 Example

Again, the algorithm was tested on the sample code from Chapter 2, and the resulting function is as follows:

$1 \to 2$
$2 \to 0$
$3 \to 3$
$4 \to 1$
$5 \to 1$
$6 \to 0$

This uses only four colours, but is still correct with respect to the graph:

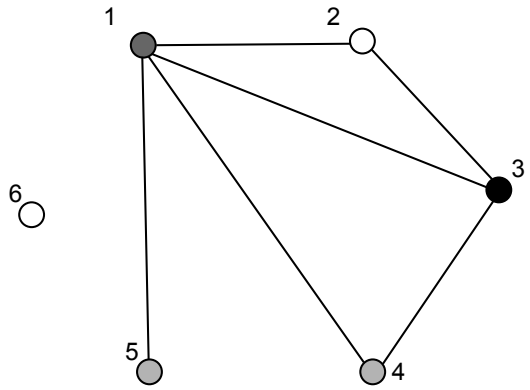

Figure 5.1: Lowest-first colouring of the sample code's clash graph

5.4 Heuristics

The two algorithms developed so far work reasonably well in practice, but can be greatly improved with extra heuristics. These heuristics are used to control the order in which vertices are considered by the colouring algorithm, so that particular vertices are prioritised and others are left as late as possible. In this section the approach to modelling heuristics is explored, and a few real heuristics are examined and verified.

5.4.1 Modelling heuristics

A convenient approach to heuristics, inspired by Kosowski & Manuszewski [7], is to break the overall algorithm into two steps. The first of these re-

orders the list of vertices in the clash graph according to some heuristic, and the second is a colouring stage such as the lowest-first algorithm verified in the last section. This means that a heuristic is correct if and only if the set of vertices after applying the heuristic is the same as the set before. Clash graphs have no duplicate vertices (this follows from the proof that the list of all registers is duplicate-free, given in Section 4.2.4), and so converting the set list into a set to test this property will not lose information. This notion is captured succinctly by the following definition:

```
heuristic_application_ok f ⟺ ∀ list. set (f list) = set list
```

5.4.2 Simple sort-based heuristics

The simplest kind of heuristic is one which simply sorts elements based on some property – for instance, a highest-degree-first ordering. This was modelled as a sort with a function passed in which computes the value of this property. The relevant definitions are as follows:

```
heuristic_insert f x [] = [x]
heuristic_insert f x (y::ys) =
if f x > f y then x::y::ys else y::heuristic_insert f x ys
heuristic_sort f [] = []
heuristic_sort f (x::xs) =
heuristic_insert f x (heuristic_sort f xs)
```

These implement a simple insertion sort, where `heuristic_insert` performs the insertion and `heuristic_sort` inserts each element into the list one by one.

To make it easier to verify heuristics based on this approach, a more specific definition of correctness was added:

```
sort_heuristic_ok f ⟺
∀ list. set (heuristic_sort f list) = set list
```

This verifies that sorting with a particular function maintains the members of the original set. This clearly implies the usual definition of heuristic cor-

rectness:

```
⊢ sort_heuristic_ok f ⇒
  heuristic_application_ok (heuristic_sort f)
```

The proof follows trivially once the definitions of `sort_heuristic_ok` and `heuristic_application_ok` are expanded.

Intuitively, if this holds for any function f then it should hold for *all* such functions:

```
⊢ sort_heuristic_ok f
```

To prove this, we first verify that `heuristic_insert` adds elements to the set correctly:

```
⊢ set (heuristic_insert f h list) = {h} ∪ set list
```

This can be proved by induction on the size of the list set to be inserted into, case splitting on whether the element to be inserted will be inserted at the current position or later. It can be used to verify that all sort-based heuristics satisfy `sort_heuristic_ok`, and therefore `heuristic_application_ok` (by the implication proved earlier).

The proof follows from the correctness of `heuristic_insert` by inducting on the size of the list and rearranging set operations using the standard HOL set theory definitions.

As a simple example, consider the following heuristic function which calculates the degree of a vertex:

```
vertex_degree (v0,[]) = 0
vertex_degree (r,x::xs) = vertex_degree (r,xs) + 1
```

Using this as the sort function with `heuristic_sort` will sort vertices according to degree, a common heuristic used when colouring graphs. Its correctness now follows trivially from the theorems proved above.

Many other sort-based heuristics are possible. For instance, if only a finite number of registers are available (a situation which is explored in Section

7.2), we might want to prioritise registers which are frequently accessed for register storage and spill those which are used less often. In this case, the clash graph generator could simply tag vertices with the number of times they are accessed in the code, and this could be used in a heuristic sort to order vertices according to how often they are accessed.

Sample run

To demonstrate that this heuristic improves generated colourings, the lowest-first colouring algorithm was run on the sample code from Chapter 2 again, this time using the highest-degree-first heuristic. The resulting function is as follows:

$1 \rightarrow 1$
$2 \rightarrow 0$
$3 \rightarrow 2$
$4 \rightarrow 0$
$5 \rightarrow 0$
$6 \rightarrow 0$

Adding the heuristic has reduced the number of colours used to only three – the minimum number of colours for the graph:

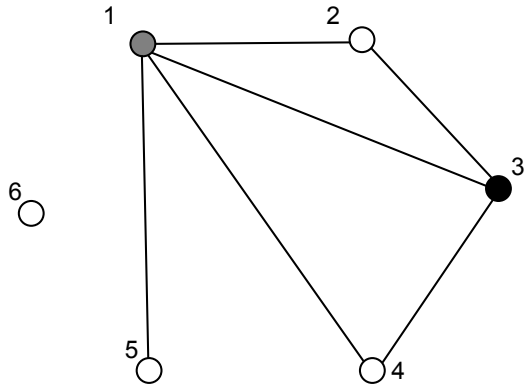

Figure 5.2: Lowest-first colouring of the sample code's clash graph, using the highest-degree-first heuristic

5.4.3 More complex heuristics

Several more elaborate heuristics are possible. The main one explored in this project is used quite frequently where there are finitely many registers available, and consists of picking at each stage the vertex with the lowest degree, removing it from the graph and placing it on a stack. This is repeated until no registers remain, and registers are then popped off the stack and coloured one at a time. The heuristic is described in Matula & Beck [9], and is very similar to a highest-degree-first heuristic such as the one in the last subsection, but with the advantage that the ordering takes into account which vertices will already be colourable at each stage. This is known to produce a better ordering than a normal highest-degree-first ordering (Kosowski & Manuszewski [7]).

This was implemented using a set of vertices which have already been considered, and writing the sort function so that it ignores them in counting the degree of a vertex. The sorting code is as follows:

```
sort_not_considered_by_degree done list =
QSORT
  (λx y.
     vertex_degree_in_subgraph done x <
     vertex_degree_in_subgraph done y ∧ ¬done (FST x)) list
```

This uses the HOL `QSORT` algorithm, passing in a predicate which states that two vertices are correctly ordered if the first one has lower degree and has not yet been considered. The predicate makes use of the following function which finds the degree of a vertex whilst ignoring those already considered (and thus removed from the graph):

```
vertex_degree_in_subgraph done (v,[]) = 0
vertex_degree_in_subgraph done (v,e::es) =
if done e then vertex_degree_in_subgraph done (v,es)
else SUC (vertex_degree_in_subgraph done (v,es))
```

The main algorithm uses an auxiliary function which iterates through a list

(assumed to be sorted by the above algorithm), putting the top element on the front of an accumulator (cs'), sorting the tail and calling itself recursively with the sorted tail and updated set of already-considered vertices (*done*):

```
lowest_degree_subgraph_heuristic_aux done [] cs' = REVERSE cs'
lowest_degree_subgraph_heuristic_aux done ((r,rs)::cs) cs' =
(let sorted = sort_not_considered_by_degree (r INSERT done) cs
 in
   lowest_degree_subgraph_heuristic_aux (r INSERT done) sorted
     ((r,rs)::cs'))
```

Due to the fact that the vertex list is sorted each time, termination of this function is non-trivial. The proof follows from the fact that each iteration shortens the length of the vertex list, as sorting maintains the length of its input. The custom induction rule generated for this function is as follows:

```
⊢ (∀ done cs'. P done [] cs') ∧
   (∀ done r rs cs cs'.
      (∀ sorted.
         (sorted =
          sort_not_considered_by_degree (r INSERT done) cs) ⇒
         P (r INSERT done) sorted ((r,rs)::cs')) ⇒
      P done ((r,rs)::cs) cs') ⇒
   ∀ v v1 v2. P v v1 v2
```

The main function calls the auxiliary with an empty 'done' set[2] and accumulator, and a sorted version of the input vertex list:

```
lowest_degree_subgraph_heuristic cs =
lowest_degree_subgraph_heuristic_aux (λ x. F)
  (sort_not_considered_by_degree (λ x. F) cs) []
```

[2]As HOL sets are represented as characteristic functions of type $\alpha \to bool$, the empty set is represented by $\lambda x.F$

Verification

The smallest-last algorithm ended up being quite non-trivial to verify. The main trick used to simplify it was the following lemma, which states that two sets formed from lists are equal when membership of one list is equivalent to membership of the other:

$\vdash$ (∀ *x* . MEM *x list* ⟺ MEM *x list′*) ⇒ (set *list* = set *list′*)

This was relatively simple to prove using the standard HOL library theorems about set equality. It was then necessary to prove that the sorting function preserves the elements passed into it:

$\vdash$ set *list* = set (sort_not_considered_by_degree *done list*)

The proof works by expanding the definition and using the above lemma along with the following standard lemma about QSORT:

$\vdash$ MEM *x* (QSORT *R L*) ⟺ MEM *x L*

The next important lemma states that anything passed in as the accumulator will end up in the final result:

$\vdash$ MEM *x acc* ⇒
MEM *x* (lowest_degree_subgraph_heuristic_aux *done list acc*)

With the custom induction rule for lowest_degree_subgraph_heuristic_aux, this was relatively simple to prove by expanding definitions and using the fact that reversing a list preserves its members (a standard HOL axiom).

The next step was to equate the set returned by the heuristic with the sets of elements passed in:

$\vdash$ MEM *x list* ∨ MEM *x acc* ⟺
MEM *x* (lowest_degree_subgraph_heuristic_aux *done list acc*)

This again used the custom induction rule. The base case was relatively simple, but the induction case required more work. The trick was to case split on whether the arbitrary x was equal to the current value at the top of the vertex list, in which case the goal follows easily by simplifying. In

the other situation, a case split was performed on whether x belongs to the accumulator. If it does, it trivially exists in the result by the lemma proved earlier. Otherwise, we case split again to see whether it belongs to the tail of the list of vertices passed in. In either case, the required goal follows by expanding definitions and using the earlier lemma stating that the sort function preserves the set of elements passed to it. Thus the proof was completed, and the heuristic was shown to be correct.

Chapter 6

Bringing everything together

In this chapter, the properties verified of the algorithms developed in previous chapters are used to produce an overall proof of correctness for the whole algorithm. The result of this is a high-level proof which shows that applying these register allocation algorithms to a piece of code will yield code with exactly the same evaluation behaviour.

This consists of three main proofs. The first shows that graphs generated by the clash graph generator possess the properties assumed by the colouring algorithms; the second shows that the definition of correctness used for colouring algorithms implies the correctness required for correct evaluation; and the third shows that a colouring satisfying the correctness used for evaluation leads to code whose evaluation behaviour is unchanged.

6.1 Linking code correctness to graph correctness

This section demonstrates that graphs generated by the clash graph generator have the necessary properties assumed by the colouring algorithms, and that a colouring deemed correct with respect to `colouring_satisfactory` will

also satisfy `colouring_ok`. As all colouring algorithms were verified using `colouring_satisfactory`, this indicates that they also satisfy `colouring_ok` and therefore that the full algorithm generates colourings satisfying `colouring_ok` for any code block.

6.1.1 Correctness of generated graphs

For the proofs about graph colouring correctness to be useful, it was necessary to show that the graphs being generated and passed in have the right well-formedness properties – recall the definitions:

```
edge_list_well_formed (v,edges) ⟺
v ∉ set edges ∧ duplicate_free edges
graph_edge_lists_well_formed es ⟺
EVERY (λx. edge_list_well_formed x) es

graph_duplicate_free [] ⟺ T
graph_duplicate_free ((r,rs)::cs) ⟺
(∀rs'. (r,rs') ∉ set cs) ∧ graph_duplicate_free cs

graph_reflects_conflicts cs ⟺
∀r1 r2 rs1 rs2.
  MEM (r1,rs1) cs ∧ MEM (r2,rs2) cs ∧ MEM r1 rs2 ⇒ MEM r2 rs1
```

We take these one at a time. Thus the first statement to be proved is as follows:

```
⊢ duplicate_free live ⇒
  graph_edge_lists_well_formed (get_conflicts code live)
```

We first prove a simpler version of this, which states that the edge list for one vertex is well-formed:

```
⊢ duplicate_free live ⇒
  edge_list_well_formed (r,conflicts_for_register r code live)
```

From the definition, we can see that this requires that a register does not appear in its own list of conflicts, and that the list of conflicts is duplicate-free. These two things were proven as lemmas in Section 4.2.4:

```
⊢ r ∉ set (conflicts_for_register r code live)
```

```
⊢ duplicate_free live ⇒
  duplicate_free (conflicts_for_register r code live)
```

These two lemmas are sufficient to prove the simplified goal above. Now that generated graph edge lists have been proven to be individually well-formed, it is relatively simple to prove the original goal:

```
⊢ duplicate_free live ⇒
  graph_edge_lists_well_formed (get_conflicts code live)
```

The proof uses one more small lemma, which relates universal quantification of a property to HOL's EVERY function:

```
⊢ (∀x. P (f x)) ⇒ EVERY P (MAP f list)
```

The proof is a trivial induction. With this, the original goal can be proved by simplifying and using the above lemma to put the statement in the form used by the simplified goal.

The next thing to prove is that a graph has no duplicate vertices. The statement is given below:

```
⊢ duplicate_free live ⇒
  graph_duplicate_free (get_conflicts code live)
```

We proved earlier on that the function `get_registers` generates duplicate-free results (see Section 4.2.4). Therefore, if the function mapping registers to edge lists preserves duplicate-freeness then the result is duplicate-free. A statement of this lemma is given below:

```
⊢ duplicate_free list ⇒
  graph_duplicate_free (MAP (λx. (x,f x)) list)
```

The proof is a simple induction where each goal can be proved by expanding definitions. With this lemma, the original goal can be proved by expanding definitions and using the fact that `get_registers` generates duplicate-free results.

The final property to be proved is `graph_reflects_conflicts`:

$\vdash$ `duplicate_free` *live* $\Rightarrow$
`graph_reflects_conflicts (get_conflicts` *code live*`)`

To begin proving this, it was necessary to show that if a register appears in another's list of conflicts then they come from a common conflicting set:

$\vdash$ `MEM` r_1 `(conflicts_for_register` r_2 *code live*`)` $\Rightarrow$
$\exists\, c$`. MEM` c `(conflicting_sets` *code live*`)` $\wedge$ `MEM` r_1 c $\wedge$ `MEM` r_2 c

This follows from the fact that for something to belong to a list union it must belong to one of the original sets. The proof that `graph_reflects_conflicts` holds is then fairly simple once definitions are expanded: both vertices must originate from the same conflicting set, and by the property below (proved in Section 4.2.4) they must therefore both appear in each other's list of conflicts.

$\vdash$ `MEM` c `(conflicting_sets` *code live*`)` $\wedge$ `MEM` r c $\wedge$ `MEM` s c $\wedge$
$r \neq s \Rightarrow$
`MEM` r `(conflicts_for_register` s *code live*`)`

This completes the proof that the generated graphs have the required properties.

6.1.2 Connecting `colouring_satisfactory` to `colouring_ok`

The next task in the proof was to show that `colouring_satisfactory`, the definition of correctness used in verifying graph colouring algorithms, implies `colouring_ok`, the definition used when working directly with code. For this it was much more convenient to use the definition `colouring_ok_alt`, which

was proved equivalent to `colouring_ok` in Section 4.2.3. The statement to be proved is given below:

```
⊢ duplicate_free live ⇒
  colouring_satisfactory c (get_conflicts code live) ⇒
  colouring_ok_alt c code live
```

This ended up being fairly complicated to prove. Initially, a slightly simpler goal was proved which would then be linked to the actual definitions to complete the original proof. The simpler goal is as follows:

```
⊢ duplicate_free live ⇒
  (∀r.
     col r ∉
     set (MAP col (conflicts_for_register r code live))) ⇒
  EVERY (λs. duplicate_free (MAP col s))
    (conflicting_sets code live)
```

This takes the usual assumption that the set of variables specified as live at the end of the code block is duplicate-free. The antecedent here states that for any register r and colouring *col*, applying the colouring to r doesn't conflict with the set of conflicts for r - the coloured r is not a member of the set of conflicts with the colouring mapped over it. As will be demonstrated formally later, this is a consequence of the colouring being satisfactory for the conflicts generated by `get_conflicts` - in fact, it is essentially a rephrasing of it. The consequent, meanwhile, states that mapping the colouring over any conflicting set yields a duplicate-free result. This is simply the definition of `colouring_respects_conflicting_sets` restated using `EVERY`, and is therefore a restatement of `colouring_ok_alt`. As will be demonstrated later, this goal is sufficient to prove the original goal.

The proof of this statement was fairly complex, making use of several of the lemmas proved earlier. The approach used was to use the following lemma from Section 4.2.3 to show duplicate-freeness of each conflicting set after colouring:

```
⊢ (∀x y.
```

x < LENGTH $list$ ∧ y < LENGTH $list$ ∧ $x \neq y$ ⇒
EL x $list$ ≠ EL y $list$) ⇒
duplicate_free $list$

To prove the antecedent of this it was necessary to use several of the lemmas about duplicate-freeness of conflicting sets along with the fact that a list being duplicate-free means no elements are equal (again from Section 4.2.3):

⊢ duplicate_free $list$ ∧ x < LENGTH $list$ ∧ y < LENGTH $list$ ∧
$x \neq y$ ⇒
EL x $list$ ≠ EL y $list$

The fact that a register's assigned colour is not assigned to any register in its conflicting set (from the antecedent of the overall goal) was then used to show that any two different registers cannot be the same after colouring, as required to show that the conflicting set after colouring is duplicate-free and complete the proof.

This is almost enough to prove the original goal. The next key lemma proves that the antecedent of the last goal is indeed a consequence of a colouring being satisfactory:

⊢ colouring_satisfactory col (get_conflicts $code$ $live$) ⇒
∀ r.
col r ∉ set (MAP col (conflicts_for_register r $code$ $live$))

To verify this it was first necessary to show that a satisfactory colouring has the desired behaviour on a single vertex:

⊢ colouring_satisfactory c cs ∧ MEM (r,rs) cs ⇒
c r ∉ set (MAP c rs)

The proof is a relatively simple induction where the goal follows from the definition of colouring_satisfactory.

Another small lemma was required, stating that if x is a member of a list then for any colouring c, cx is a member of the list after mapping c over it:

⊢ MEM x xs ⇒ MEM (c x) (MAP c xs)

The proof is trivial using simplification and evaluation, along with the HOL theorem `MEM_MAP`:

$\vdash$ `MEM` x `(MAP` f l`)` $\iff$ $\exists y$`. (`x `=` f y`)` $\wedge$ `MEM` y l

Now the goal `colouring_satisfactory_expansion` can be verified. This makes use of the following lemma which was given in Section 4.2.4, to avoid having to restrict the statement to registers which are used in the program:

$\vdash$ $r \notin$ `set (get_registers` *code live*`)` $\Rightarrow$
`(conflicts_for_register` r *code live* `= [])`

The proof works in essence by expanding out the definition of colouring_satisfactory showing that it means any r must satisfy the goal of the statement, but this was fairly complex to prove due to the complexity of colouring_satisfactory and used several of the lemmas proved above.

Thus we have verified that the antecedent of the overall goal implies the antecedent of `respecting_register_conflicts_respects_conflicing_sets`. The full proof can now be derived using these lemmas, expanding the definition of `colouring_ok_alt` to make the goal match up to the consequent of `respecting_register_conflicts_respects_conflicting_sets`.

Therefore all colourings which satisfy colouring_satisfactory will also satisfy `colouring_ok_alt` and thus `colouring_ok`. This gives a full end-to-end proof showing that the full register allocation algorithm generates a colouring which satisfies `colouring_ok`. The remaining step in proving the project's overall goal is to show that a colouring satisfying `colouring_ok` will not affect the behaviour of a block of code, and the proof of this is explained in the next section.

6.2 Overall proof of correctness

Now that the algorithm has been proven to generate colourings satisfying `colouring_ok`, it remains to show that `colouring_ok` is sufficient to prove that the code behaviour will be unchanged. This statement is as follows:

```
⊢ colouring_ok c code live ∧ no_dead_code code live ∧
  (MAP s (get_live code live) =
   MAP (t o c) (get_live code live)) ⇒
  (MAP (eval f s code) live =
   MAP (eval f t (apply c code) o c) live)
```

6.2.1 The statement of correctness

There are three main components to the antecedent:

- The colouring c satisfies `colouring_ok`
- There is no dead code – this will be discussed shortly, but essentially states that there aren't any instructions which modify non-live registers
- The two stores s and t agree on the live variables. The second store t is the store after register allocation, hence it is being composed with the colouring c in the statement of store equality

The goal to be proved claims the equality of the store with and without colouring after the code has been evaluated. On the left side of the equality, the code is evaluated and the resulting store is mapped over the variables specified as live after the code has executed to get the results. On the right, the colouring is applied to the code and it is then evaluated with the store t (s after register allocation). This is then mapped over the coloured live variables by composing it with c to obtain the equivalent results after colouring. The equality of these indicates that the results of evaluating the code are exactly the same before and after register allocation.

6.2.2 Dead code elimination

Before proving the correctness statement, it was necessary to define a function to eliminate dead code and verify its correctness with `no_dead_code`. The definition of `no_dead_code` is given below:

```
no_dead_code [] v0 ⟺ T
no_dead_code (Inst w r1 r2::code) live ⟺
MEM w (get_live code live) ∧ no_dead_code code live
```

The dead code removal function was fairly simple to define, and is given below:

```
remove_dead_code [] live = []
remove_dead_code (Inst w r1 r2::code) live =
(let newcode = remove_dead_code code live
 in
   if MEM w (get_live newcode live) then Inst w r1 r2::newcode
   else newcode)
```

The proof of correctness was relatively easy, using induction and case splitting on whether the current instruction is dead. The correctness statement is as follows:

```
⊢ no_dead_code (remove_dead_code code live) live
```

6.2.3 Lemmas

The correctness proof depends on a number of lemmas. Firstly, the proof was by induction and so making use of the inductive hypothesis required a pair of proofs showing that `colouring_ok` and `no_dead_code` are preserved when the first instruction is removed. The proofs of these statements were trivial by expanding definitions:

$$\frac{\texttt{colouring\_ok } c \texttt{ (Inst } n \ n_0 \ n_1\texttt{::}code\texttt{) } live}{\texttt{colouring\_ok } c \ code \ live}$$

$$\frac{\texttt{no\_dead\_code (Inst } n \ n_0 \ n_1\texttt{::}code\texttt{) } live}{\texttt{no\_dead\_code } code \ live}$$

Another small yet useful lemma is the following, stating that `colouring_ok` means applying the colouring to the current set of live variables yields a duplicate-free result:

```
          colouring_ok c code live
-------------------------------------------------
duplicate_free (MAP c (get_live code live))
```

This was proved by case splitting on the code variable then evaluating and simplifying the result to prove the goal.

It was also useful to be able to state that if the result of mapping a colouring over a list is duplicate-free and one considers two unequal elements of the original list, they will still not be equal after applying the function to each of them:

```
⊢ duplicate_free (MAP c live) ∧ x ≠ y ∧ MEM x live ∧
  MEM y live ⇒
  c x ≠ c y
```

This was proved in Section 4.2.3.

These last two lemmas make it possible to prove the following statement, which essentially states that a colouring satisfying `colouring_ok` for a particular block of code is injective on the set of live variables for that code:

```
⊢ no_dead_code code live ∧ colouring_ok c code live ∧ x ≠ y ∧
  MEM x (get_live code live) ∧ MEM y (get_live code live) ⇒
  c x ≠ c y
```

This follows trivially from the two lemmas above.

6.2.4 Proof of correctness

It is now possible to complete the proof of the original statement of correctness:

```
⊢ colouring_ok c code live ∧ no_dead_code code live ∧
  (MAP s (get_live code live) =
   MAP (t o c) (get_live code live)) ⇒
  (MAP (eval f s code) live =
   MAP (eval f t (apply c code) o c) live)
```

The proof is by induction on the block of code, and uses all the lemmas above along with various definition expansions and some of the HOL lemmas regarding `MAP` and `MEM`. With this proved, it follows that the full register allocation algorithm is correct in that it doesn't affect code behaviour with respect to evaluation.

Chapter 7

Extension features

Introduction

The algorithms verified so far form a complete functional register allocator. However, the allocator disregards some important practical considerations. This chapter covers the extension features which were implemented and verified after the core algorithms had been proved correct, and discusses how the system might be further extended to allow for features not considered in this project.

The first extension is the addition of preference graphs to influence colouring choices. In real systems, there are many situations where a value has to be moved into a particular register before it is used: a particular operator may require its inputs to be placed in certain registers, say, or it may output to a temporary 'result' register meaning the result has to be moved away to avoid being clobbered by the next operation. The processor could also have a particular convention for procedure calls which requires certain values to be stored in appropriate registers. Having lots of move instructions around certain operations wastes time, and so a good optimising compiler will make an effort to remove these where possible.

Preference graphs are a solution to this: they link virtual registers which

should ideally be mapped to the same physical register, making a move instruction redundant and allowing it to be safely removed. An algorithm was verified which generates such a preference graph, and a modified colouring algorithm was written to use these preferences wherever possible. A function to remove redundant moves after colouring was then implemented and proven correct.

Another important consideration is the lack of infinite registers in real systems. This requires certain values to be spilled to memory, and as a result instructions have to be added to the code to load and store these spilled values. This extension largely amounted to generating these instructions and proving that the resulting behaviour is the same, and possible heuristics for the situation where some registers will be spilled are discussed.

7.1 Preference graphs

7.1.1 Representation

A preference graph is broadly similar to a clash graph: each register has an adjacency list of other registers, in this case those with which it should ideally share a physical register. However, to combine this graph with the clash graph effectively it was necessary to use a structure allowing an arbitrary register to be looked up without iterating through the structure. Furthermore, preference graphs are never iterated through in the same way as clash graphs, and so a list of adjacency lists was deemed inappropriate in this case.

Instead, a preference graph is a function mapping registers to lists of 'preferred' registers, initialised to $\lambda x.[]$ for registers with no preferences. This makes it much easier for the algorithm to look up a register's preferences quickly.

7.1.2 Generating preference graphs

Move instructions

To make it possible to generate preference graphs, it was necessary to extend the code to include 'move' instructions. For this purpose, the code definition was extended to include an operation as well as the three registers:

```
instr = Instr of op ⇒ num ⇒ num ⇒ num
```

The 'op' parameter is either an arbitrary function of type $num \rightarrow num \rightarrow num$ or a move, as defined below:[1]

```
op = Op of num -> num -> num | Move
```

An auxiliary function was implemented to evaluate the action of an operation:

```
eval_op (Op f) = f
eval_op Move = (λ x1 x2. x1)
```

In the case of a `Move` instruction, it returns a function which simply returns the first argument register, to be stored in the destination register. The evaluation function is then re-defined appropriately, using `eval_op`:

```
eval_instr s [] = s
eval_instr s (Instr op w r1 r2::code) =
(let f = eval_op op
 in
   eval_instr ((w =+ f (s r1) (s r2)) s) code)
```

Building graphs

A preference graph simply contains edges between those vertices which are source and destination in a *move* instruction. The function to build a graph

[1] Again, this uses the HOL datatype definition syntax. In this case there are two constructors: `Op`, taking three `num` values representing registers, and `Move`, taking two `num`s representing source and destination.

must therefore scan the code for *move* instructions and add an edge whenever one is found:

```
compute_preferences [] = (λ x. [])
compute_preferences (Instr Move dest source v0::code) =
add_preference_pair (source,dest) (compute_preferences code)
compute_preferences (Instr (Op v11) v4 v5 v6::code) =
compute_preferences code
```

The function `add_preference_pair` simply adds each of the registers to the other's list of preferences by updating the preferences function:

```
add_preference_pair (x,y) prefs =
(let xs = prefs x in
 let ys = prefs y
 in
   (x =+ y::xs) ((y =+ x::ys) prefs))
```

7.1.3 Altered lowest-first colouring algorithm

To take these preferences into account, an alternative colouring function was implemented taking a preference graph as an extra argument:

```
lowest_first_preference_colouring [] prefs = (λ x. 0)
lowest_first_preference_colouring ((r,rs)::cs) prefs =
(let col = lowest_first_preference_colouring cs prefs in
 let lowest_available = lowest_available_colour col rs in
 let preference_colour =
       best_preference_colour col rs (prefs r)
 in
   if preference_colour ∉ set (MAP col rs) then
     (r =+ preference_colour) col
   else (r =+ lowest_available) col)
```

This uses the auxiliary function `best_preference_colour`:

```
best_preference_colour col rs [] = 0
```

```
best_preference_colour col rs (p::ps) =
if col p ∉ set rs then col p
else best_preference_colour col rs ps
```

The `best_preference_colour` function iterates through preferences until it finds one which can be satisfied, returning zero if none can be satisfied. The main function takes the result of this for the current vertex and sees whether it conflicts – if it does, normal lowest-first colouring is used; if it does not, the preference colour is used.

Correctness

The correctness of this function is stated below:

```
⊢ graph_reflects_conflicts cs ∧ graph_duplicate_free cs ∧
  graph_edge_lists_well_formed cs ⇒
  colouring_satisfactory
    (lowest_first_preference_colouring cs prefs) cs
```

This proof was very similar to the proof of lowest-first colouring's correctness (see Section 5.3.2), but with a case split on whether or not the preference colour was used. If it was, then the 'if' statement in the function guarantees that it doesn't conflict with the current register's neighbours. Otherwise, the lowest-first colour is used which can be shown to be safe in the same way as it was shown for plain lowest-first colouring.

7.1.4 Removing redundant moves

With a preference-driven colouring obtained, the next step is to remove any 'move' instructions which have become redundant after colouring. A function to handle this was implemented:

```
remove_identity_moves [] = []
remove_identity_moves (Instr Move d s v::code) =
if d = s then remove_identity_moves code
```

else `Instr Move` d s v`::remove_identity_moves` *code*
`remove_identity_moves (Instr (Op` v_{11}`)` v_4 v_5 v_6`::`*code*`) =`
`Instr (Op` v_{11}`)` v_4 v_5 v_6`::remove_identity_moves` *code*

This leaves the code unchanged unless it finds a 'move' instruction with equal source and destination registers, in which case it drops the instruction. It was necessary to prove that this doesn't affect evaluation behaviour of the code:

$\vdash$ `eval_instr` s *code* `=`
  `eval_instr` s `(remove_identity_moves` *code*`)`

The proof of this is by induction, with a trivial base case. The inductive case follows by taking cases on the instruction type and the operator, with the only non-trivial case being a 'move' instruction. Case-splitting on equality of source and destination gives two trivial goals which can be proved immediately by evaluation and simplification.

7.1.5 Example

Consider the following small piece of code, which features two moves followed by an addition:

```
R1 = R2
R4 = R3
R5 = R1 + R4
```

The graph for this, including generated preferences and excluding the output register R_5, is given in Figure 7.1. Preference edges are displayed as dashed lines. Applying the lowest-first colouring algorithm with preferences generated by `compute_preferences` yields the colouring in Figure 7.2.

Here, all the preferences in the code have been satisfied. Applying the colouring and removing redundant moves gives the following output (where R_5 has been assigned to register zero because it doesn't conflict with anything):

```
R0 = R0 + R1
```

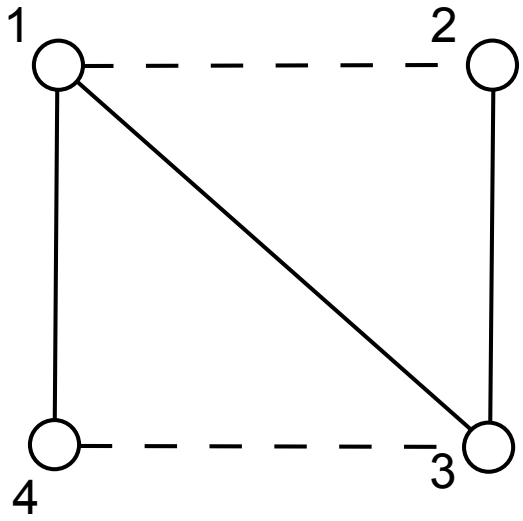

Figure 7.1: Clash graph with preferences

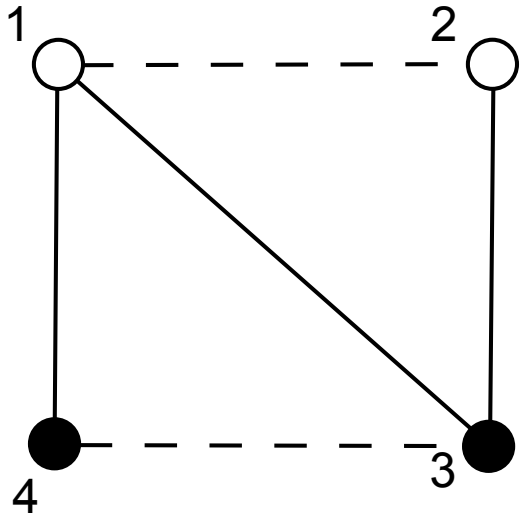

Figure 7.2: Coloured clash graph with preferences

7.2 Finite registers and spilling

7.2.1 Approach

The simplest approach to modelling spilling in this case was to specify a maximum number of registers K and say that all allocated registers above

that number are spilled. This leaves the colouring algorithms as they are, and the only change to be made is to add a phase which generates loads and stores in the target code.

To model this, two 'stores' were used instead of one. The first represents registers, and the second represents memory. Locations above or equal to K are loaded from the memory store and those lower are loaded from the register store. Three registers – K, $K+1$ and $K+2$ – are set aside to be used as temporary register storage for the result and the two arguments, where needed. In practice, one of the registers could be re-used to store the result of the operation, but three were used here to simplify verification. The code datatype was also updated to include load and store instructions:

```
spill_inst
  = Load of
  num ⇒ num
  | Store of
  num ⇒ num
  | ThreeAddr of
  num ⇒ num ⇒ num
```

Instructions before this stage are assumed never to contain loads or stores, and so continue using the original code datatype. The following simple function converts normal code into code with loads and stores:

```
to_spill_inst [] = []
to_spill_inst (Inst w r1 r2::code) =
ThreeAddr w r1 r2::to_spill_inst code
```

A new evaluation function was then defined for this augmented instruction type:

```
eval_stack f (r,s) K [] = merge (r,s) K
eval_stack f (r,s) K (Load reg mem::code) =
eval_stack f ((reg =+ s mem) r,s) K code
eval_stack f (r,s) K (Store mem reg::code) =
eval_stack f (r,(mem =+ r reg) s) K code
```

```
eval_stack f (r,s) K (ThreeAddr w r1 r2::code) =
eval_stack f ((w =+ f (r r1) (r r2)) r,s) K code
```

The `merge` function is used to combine memory and registers into one function, and is described in the next section.

A pair of functions was then defined to generate loads and stores for spilled registers:

```
spill_stores K [] = []
spill_stores K (ThreeAddr w r1 r2::code) =
if w ≥ K then ThreeAddr K r1 r2::Store w K::spill_stores K code
else ThreeAddr w r1 r2::spill_stores K code
spill_stores K (Load d s::code) = Load d s::spill_stores K code
spill_stores K (Store d s::code) =
Store d s::spill_stores K code

spill_loads K [] = []
spill_loads K (ThreeAddr w r1 r2::code) =
if r1 ≥ K ∧ r2 ≥ K then
  Load (K + 1) r1::Load (K + 2) r2::
      ThreeAddr w (K + 1) (K + 2)::spill_loads K code
else if r1 ≥ K then
  Load (K + 1) r1::ThreeAddr w (K + 1) r2::spill_loads K code
else if r2 ≥ K then
  Load (K + 2) r2::ThreeAddr w r1 (K + 2)::spill_loads K code
else ThreeAddr w r1 r2::spill_loads K code
spill_loads K (Load d s::code) = Load d s::spill_loads K code
spill_loads K (Store d s::code) = Store d s::spill_loads K code
```

The following function handles all instruction generation using the two functions:

```
spill_loads_stores K code = spill_stores K (spill_loads K code)
```

7.2.2 Verification

Verification of this step consisted of proving that code with loads and stores has the same behaviour as code without. This was demonstrated independently for the cases where a result is stored in memory and for where a necessary argument is stored in memory, and was verified over execution of one arbitrary instruction (assuming the stores are equal before) rather than inductively on a block of code. This is trivially equivalent, but using this approach made the proof far clearer.

A key part of the proof is the way equality of a purely register-based store and a store combining registers and memory is defined. As mentioned, the latter involves two stores, and so functions for converting between the two were defined:

```
split r K =
((λx. if x < K then r x else 0),(λx. if x ≥ K then r x else 0))

merge (r,s) K = (λx. if x ≥ K then s x else r x)
```

The correctness of these two follows from the theorem below, which was proved using evaluation and expanding definitions:

```
⊢ merge (split s K) K = s
```

Furthermore, the following theorem demonstrates that `split` is correct in that it is equal to the original store on all values:

```
⊢ (split s K = (r,m)) ⇒
  (∀x. x ≥ K ⇒ (m x = s x)) ∧ ∀x. x < K ⇒ (r x = s x)
```

This was again proved using evaluation and simplification.

With these two functions proved correct, the correctness of generated stores was then verified:

```
⊢ r1 < K ∧ r2 < K ⇒
  (eval f s [Inst w r1 r2] =
   eval_stack f (split s K) K
```

```
    (spill_stores K (to_spill_inst [Inst w r1 r2])))
```

Note that this assumes that the stores used in both evaluations are equal with respect to the definition of split. It also assumes that the other two registers used are stored in registers, to simplify the proof. The proof then follows by expanding all definitions and demonstrating equality of the resulting stores by case splitting and using the following HOL axiom:

```
⊢ (∀x. f x = g x) ⇒ (f = g)
```

The proof of correctness where one of the arguments has been spilled was very similar. The statement of correctness is given below:

```
⊢ w < K ∧ r2 < K ∧ ((r,m) = split s K) ⇒
  (eval f s [Inst w r1 r2] =
   eval_stack f (r,m) K
     (spill_loads K (to_spill_inst [Inst w r1 r2])))
```

The proof for the other argument is near-identical.

7.2.3 Heuristics for spilling

A number of heuristics can be used to make sure the registers being spilled have as minimal an impact on performance as possible. For instance, one could spill the register which conflicts with the most others (to ensure fewer registers have to be spilled). Another would be to try and find registers which contain values computed once at the start of their live range but not used until many instructions later, as these would be prime candidates for storing in memory.

For this project, a relatively common one was verified: spill the virtual register which has the fewest uses. To implement this, a function was written which counts the number of uses of each register and creates a mapping from registers to uses. This function is given below:

```
count_uses r code =
LENGTH (FILTER (λx. x = r) (flatten_used_registers code))
```

The function simply takes a list of all registers used in instructions, filters them to instances of the register in question and returns the length. It relies on the following auxiliary function which generates the list of used registers:

```
flatten_used_registers [] = []
flatten_used_registers (Inst w r1 r2::code) =
w::r1::r2::flatten_used_registers code
```

The following function takes the information from `count_uses` and uses it to create a mapping from registers to frequencies:

```
get_uses_function code = (λ r. count_uses r code)
```

A heuristic was then implemented which takes this function as well as the list of edge lists and sorts registers from least- to most-used:

```
most_used_last_heuristic uses list =
QSORT (λ x y. uses x < uses y) list
```

As colouring works from the back of the list forwards, this will result in frequently-used registers being coloured first and thus being less likely to be spilled. The correctness of this function follows trivially from the fact that `QSORT` maintains the set passed into it:

```
⊢ heuristic_application_ok (most_used_last_heuristic uses)
```

7.2.4 Example

Consider again the code given in Chapter 2:

```
R1 = R2 + R3
R4 = R1 * R2
R5 = R3 - R4
R6 = R1 + R5
```

Now, assume there are only six registers available on the system, numbered from zero upwards. All registers higher than two must therefore be spilled, with 3, 4 and 5 allocated as temporary storage for arguments and results.

Using the functions described in this section, we obtain the following code (where R_n signifies register n and plain integers refer to memory locations):

```
Load R5 3
R1 = R2 + R5
R3 = R1 * R2
Store 4 R3
Load R4 3
Load R5 4
R3 = R4 - R5
Store 5 R3
Load R5 5
R3 = R1 + R5
Store 6 R3
```

Chapter 8

Conclusions

8.1 Evaluation

Overall, the project was successful in its original aims. The system implemented has a good colouring algorithm and a series of effective heuristics, and a complete proof of correctness showing that the code emitted has the same evaluation behaviour as the input code. It also successfully models preference graphs, with a verified colouring algorithm which takes them into account, and register spilling, with a verified code generation step and heuristic. It is also fairly modular, and as was demonstrated in Chapter 7 it is fairly easy to extend and add extra heuristics and features without too much extra verification effort.

The main weakness is that the code representation is fairly limited – for instance, it doesn't allow for jumps. This is arguably not significant, as the algorithm could easily be run on one basic block at a time, but for register mappings to be maintained between blocks the algorithm needs to be extended to handle this. This is discussed further in the next section on future work.

Another problem is performance: the algorithm as implemented is not at all optimised, and would likely be quite slow compared to most real allocators.

However, this problem is largely orthogonal to the objectives of the project as the intent was to verify the algorithms rather than implement them in the most efficient way, and any faster implementation satisfying the necessary properties could easily be substituted in.

It therefore seems reasonable to consider the project a success. With that said, there are a number of areas where improvements could be made. These are discussed in the next section.

8.2 Future work

The last section highlighted a couple of issues with the algorithms implemented. In this section, solutions to these problems are examined along with other possible improvements which could be made.

8.2.1 Improved code representation

The first main issue is that the code datatype is very limiting: all instructions are simply three-address operations, meaning there can be no jumps or procedure calls. An interesting improvement would be to extend the code representation to allow for these instructions. Such an implementation would likely require that the code be split into basic blocks, and then the standard live variable analysis algorithm could be run on the result. This wouldn't be particularly difficult, and would only change the implementation and verification of the `get_live` function – most other proofs will still stand as they are. A new evaluation function would be required, however, and so correctness of the generated code would have to be re-proved with respect to it.

8.2.2 Better colouring algorithms

The implemented colouring algorithms are good, but more intelligent colouring algorithms are available. For instance, the iterated register coalsecing

algorithm described in George [5] would be a good candidate. Again, this wouldn't affect many other things, but would potentially require some re-structuring of the graph colouring proofs.

8.2.3 More thorough treatment of register spilling

While fairly successful, the extension work on finite registers and spilling is somewhat simplistic at the moment. For instance, in a real compiler one wouldn't allocate three registers for temporary variables loaded from memory as this would be wasteful. Instead, the new temporaries are generally added into the original set of registers and register allocation is performed again on the new code with loads and stores.

Furthermore, Patterson [11] mentions some better approaches to spilling, such as breaking up live ranges, and it would likely be useful to implement and verify some of these as well.

8.2.4 Performance considerations

The algorithms implemented are clearly not very optimised. An interesting extension to the project would be re-implementing some of the key functions in more efficient ways, and verifying that they implement the same behaviour as the original unoptimised versions. As mentioned, the modular nature of the proofs means this wouldn't be particularly hard, and the resulting system would be more practically useful.

Bibliography

[1] Hussein Al-Omari and Khair Eddin Sabri. New graph coloring algorithms. *Journal of Mathematics & Statistics*, 2(4), 2006.

[2] Gertrud Bauer and Tobias Nipkow. The 5 colour theorem in isabelle/isar. In *THEOREM PROVING IN HIGHER ORDER LOGICS, VOLUME 2410 OF LNCS*, pages 67–82. Springer-Verlag, 2002.

[3] Sandrine Blazy, Benot Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation, 2010.

[4] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *J. ACM*, 23(1):43–49, January 1976.

[5] Lal George. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18:300–324, 1996.

[6] HeuristicsWiki. Degree based ordering. `http://heuristicswiki.wikispaces.com/Degree+based+ordering`.

[7] Adrian Kosowski and Krzysztof Manuszewski. Classical coloring of graphs. *Graph colorings*, 352, 2004.

[8] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009.

[9] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

[10] Lars Noschinski. A Graph Library for Isabelle. 2014.

[11] Jason Robert Carey Patterson. Register allocation by graph colouring. `http://www.lighterra.com/papers/graphcoloring/`, 2003.

[12] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling.

[13] Joseph Tassarotti. Register allocation in CompCert. http://gallium.inria.fr/blog/register-allocation/, August 2012.

[14] Eric Weisstein. Vertex coloring. http://mathworld.wolfram.com/VertexColoring.html.