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 the degree of Master of Philosophy in Advanced Computer Science

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

Email: David.Barker@cl.cam.ac.uk

May 17, 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 is the abstract. Write a summary of the whole thing. Make sure it fits in one page. TODO.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is the introduction where you should introduce your work. In general the thing to aim for here is to describe a little bit of the context for your work — why did you do it (motivation), what was the hoped-for outcome (aims) — as well as trying to give a brief overview of what you actually did.

It's often useful to bring forward some "highlights" into this chapter (e.g. some particularly compelling results, or a particularly interesting finding).

It's also traditional to give an outline of the rest of the document, although without care this can appear formulaic and tedious. Your call.

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 have only finite numbers 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 [ref?]. Because this is difficult to decide with absolute precision, we gener-

ally 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 (TODO ref). 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.

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. 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 (TODO section)), 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 (TODO ref?).

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 (TODO ref).

A more complex heuristic, which is useful when attempting to colour with at most K colours, is to pick at each stage the vertex of highest degree less than K (this constraint on its degree ensures that it can definitely be coloured if only K colours are available). 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 lowest 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

TODO

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 projects overall goal will follow in Chapter (TODO: bringing everything together).

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 trivially be 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:

```
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 (TODO: extensions chapter).

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 (TODO: extensions-complex code). The main difference in using this representation is that live variable analysis is somewhat simpler.

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 (TODO bringing together - proving stuff about code evaluation), where it is proven to hold for the register allocator built during this project. The evaluation function used is given below:

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

Here f is an arbitrary binary operator and s is the store, a total function representing the values of all registers. 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 (TODO: LVA-colouring_ok).

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
```

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 (TODO: LVA-colouring_-ok_def) 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 HOLs 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 (TODO: colouring_ok_def - duplicate_free stuff).)

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 induction.

```
⊢ 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.

```
⊢ MEM x xs ⇒ MEM x (insert y xs)
```

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

```
⊢ a ∉ set (insert x xs) ⇒ a ∉ 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.

$\vdash$ `MEM` x $list$ $\vee$ `MEM` x $list'$ $\Rightarrow$ `MEM` x `(list_union` $list$ $list'$`)`

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:

$\vdash$ `MEM` $list$ $lists$ $\wedge$ `MEM` x $list$ $\Rightarrow$
`MEM` x `(list_union_flatten` $lists$`)`

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 didnt 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 (TODO background bit), 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:

```
⊢ (MAP s (get_live code live) = MAP t (get_live code live)) ⇒
  (MAP (eval f s code) live = MAP (eval f t code) 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 programs evaluation will agree on whatever variables are designated as live at the end. The proof of this statement is fairly complex, and is (TODO write up proof).

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:

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

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

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

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 (delete` *n list*`)` $\iff$ `duplicate_free` *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.

$\vdash$ `EVERY (`λ *list*`. duplicate_free` *list*`)` *lists* $\Rightarrow$
`duplicate_free (list_union_flatten` *lists*`)`

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 duplicate_free_insertion and duplicate_free_-deletion and the definition of get_live:

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

Finally, the following useful property was proved:

$\vdash$ `duplicate_free (MAP` *c live*`)` $\wedge$ $x \neq y$ $\wedge$ `MEM` *x live* $\wedge$
`MEM` *y live* $\Rightarrow$
$c\ x \neq c\ y$

This states 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. 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:

`colouring_ok` *c* `[]` *live* $\iff$ `duplicate_free (MAP` *c live*`)`
`colouring_ok` *c* `(Inst` *w* r_1 r_2`::`*code*`)` *live* $\iff$
`duplicate_free (MAP` *c* `(get_live (Inst` *w* r_1 r_2`::`*code*`)` *live*`))` $\wedge$
`colouring_ok` *c code live*

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`.

$\vdash$ `colouring_ok` *c* `(Inst` *n* n_0 n_1`::`*code*`)` *live* $\Rightarrow$
`colouring_ok` *c code live*

This definition works nicely for the proofs involving 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 (TODO bringing together - satisfactory implies ok).

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 HOLs `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 (TODO extensions).

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

$\vdash$ `colouring_ok_alt` *c code live* $\iff$ `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 (λ set. 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 registers 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:

```
⊢ duplicate_free live ⇒
  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 others list of conflicts. These theorems are revisited later in Section (TODO bringing it all together), 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:

$$\begin{array}{l}\vdash r \notin \texttt{set (get\_registers } code\ live\texttt{)} \wedge \\ \quad \texttt{MEM (Inst } w\ r_1\ r_2\texttt{)}\ code \Rightarrow \\ \quad r \neq w \wedge r \neq r_1 \wedge r \neq r_2\end{array}$$

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:

$$\begin{array}{l}\vdash r \notin \texttt{set (get\_registers } code\ live\texttt{)} \Rightarrow \\ \quad r \notin \texttt{set (get\_live } code\ live\texttt{)}\end{array}$$

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:

$$\begin{array}{l}\vdash r \notin \texttt{set (get\_registers } code\ live\texttt{)} \wedge \\ \quad \texttt{MEM } set\ \texttt{(conflicting\_sets } code\ live\texttt{)} \Rightarrow \\ \quad r \notin \texttt{set } set\end{array}$$

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 dont get used in the code. This useful propertyis given below:

$$\begin{array}{l}\vdash r \notin \texttt{set (get\_registers } code\ live\texttt{)} \Rightarrow \\ \quad \texttt{(conflicts\_for\_register } r\ code\ live\ \texttt{= [])}\end{array}$$

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 $\wedge$
$r \neq s \Rightarrow$
`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* $\Rightarrow$
`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 Section (TODO bringing it all together).

Chapter 5

Colouring algorithms

TODO

Chapter 6

Bringing everything together

TODO

Chapter 7

Extension features

TODO

Chapter 8

Summary and conclusions

TODO