

# Developing a formally verified algorithm for register allocation

A Part III project

David Barker

9<sup>th</sup> June 2014

# Introduction

## The problem of register allocation

- Intermediate code assumes infinite registers, whilst real machines have finite registers
- Using memory costs many cycles
- When two variables are never used at the same time they can be stored in the same register
- Minimising register usage allows more data to be stored in registers

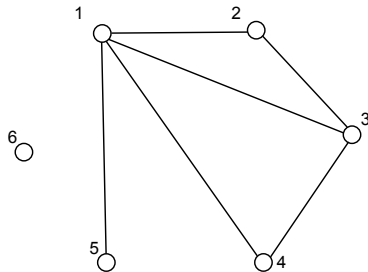
# Register allocation by graph colouring

## Computing live ranges

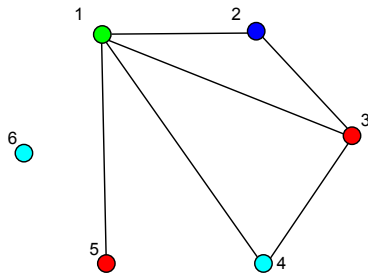
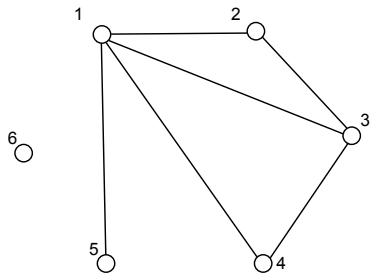
$R1 = R2 + R3 \quad \{R_2, R_3\}$   
 $R4 = R1 * R2 \quad \{R_1, R_2, R_3\}$   
 $R5 = R3 - R4 \quad \{R_1, R_3, R_4\}$   
 $R6 = R1 + R5 \quad \{R_1, R_5\}$

# Building a clash graph

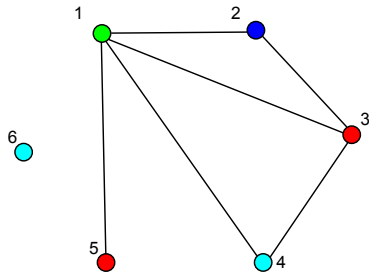
$R1 = R2 + R3 \quad \{R_2, R_3\}$   
 $R4 = R1 * R2 \quad \{R_1, R_2, R_3\}$   
 $R5 = R3 - R4 \quad \{R_1, R_3, R_4\}$   
 $R6 = R1 + R5 \quad \{R_1, R_5\}$



# Colouring the clash graph



# Applying the colouring



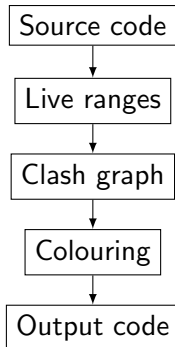
$$R1 = R2 + R3$$

$$R4 = R1 * R2$$

$$R3 = R3 - R4$$

$$R4 = R1 + R3$$

# The full algorithm



A correct algorithm will generate output code with exactly the same behaviour

# How we ensure this behaviour

A correct algorithm produces a colouring which causes no conflicts between simultaneously live registers:

```
colouring_ok_alt c code live  $\iff$   
colouring_respects_conflicting_sets c  
  (conflicting_sets code live)
```

This was proved sufficient: a colouring satisfying this will always yield code with unchanged behaviour



# Code representation

A block of code is represented by a list of three-address instructions:

`inst = Inst of num  $\Rightarrow$  num  $\Rightarrow$  num`

This is evaluated on a store  $s$  as follows:

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

Colourings are functions of type  $num \rightarrow num$

Colourings can be applied simply by substituting registers:

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

# Set representation

To simplify definitions and proofs, sets are represented as duplicate-free lists and all functions manipulating them are proven to preserve duplicate-freeness

Many simple set functions were implemented preserving this representation, for example:

```
insert x xs = if MEM x xs then xs else x::xs
```

```
delete x xs = FILTER ( $\lambda y. x \neq y$ ) xs
```

# The algorithm

## Live variable analysis

The set of live variables before a block of code is given by the following equation:

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

This was implemented as follows:

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

# Correctness

This was implicitly proved correct as its usage led to an algorithm proven to generate behaviour-preserving colourings

More directly, it was proved that only registers returned by `get_live` affect program behaviour:

$$\vdash (\text{MAP } s \text{ (get\_live code live)} = \text{MAP } t \text{ (get\_live code live)}) \Rightarrow \\ (\text{MAP (eval } f \text{ s code) live} = \text{MAP (eval } f \text{ t code) live})$$

# Clash graph generation

## Clash graph representation

Graphs are represented as lists of (vertex, clash list) pairs, for example:

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

Here  $r_n$  is the  $n^{th}$  register and  $c_n$  is the  $n^{th}$  register conflicting with it.

This makes it simple to iterate over vertices, and the list can be re-ordered to prioritise certain vertices for colouring.

# Building the graph

First we need to get the list of registers conflicting with a given register:

```
conflicts_for_register r code live =  
  delete r  
    (list_union_flatten  
      (FILTER ( $\lambda$ set. MEM r set) (conflicting_sets code live)))
```

This function is then used to build a graph in the specified format:

```
get_conflicts code live =  
  MAP ( $\lambda$ reg. (reg, conflicts_for_register reg code live))  
    (get_registers code live)
```

# Correctness of generated clash graphs

Verification of the clash graph generation stage consisted of three main proofs:

- Registers never conflict with themselves (follows easily from the definition of `conflicts_for_register`)  
$$\vdash r \notin \text{set } (\text{conflicts\_for\_register } r \text{ code live})$$
- The graph is complete: any registers from the same conflicting set appear in each other's conflicts  
$$\vdash \text{MEM } c \text{ (conflicting\_sets code live)} \wedge \text{MEM } r \text{ } c \wedge \text{MEM } s \text{ } c \wedge r \neq s \Rightarrow \text{MEM } r \text{ (conflicts\_for\_register } s \text{ code live)}$$



- The graph doesn't contain any false conflicts: every conflict is the result of two registers appearing in a conflicting set together

$$\vdash \text{MEM } r_1 \text{ (conflicts\_for\_register } r_2 \text{ code live)} \Rightarrow \\ \exists c. \text{MEM } c \text{ (conflicting\_sets code live)} \wedge \text{MEM } r_1 \text{ } c \wedge \text{MEM } r_2 \text{ } c$$

# Colouring algorithms

## Defining correctness

A graph colouring is correct if no vertex has the same colour as any of its neighbours. This is captured in the definition below:

$$\begin{aligned} \text{colouring\_satisfactory } col \ [] &\iff T \\ \text{colouring\_satisfactory } col \ ((r,rs)::cs) &\iff \\ col \ r \notin \text{set } (\text{MAP } col \ rs) \wedge \text{colouring\_satisfactory } col \ cs \end{aligned}$$

This was shown to imply the earlier definition of colouring correctness:

$$\begin{aligned} \vdash \text{duplicate\_free } live &\Rightarrow \\ \text{colouring\_satisfactory } c \ (\text{get\_conflicts } code \ live) &\Rightarrow \\ \text{colouring\_ok\_alt } c \ code \ live \end{aligned}$$

Thus proving that a colouring satisfies `colouring_satisfactory` is sufficient to show that it preserves program behaviour

# Requirements on clash graphs

For verification to work, it was necessary to show that generated graphs satisfy several properties:

- Edge lists must contain no duplicates and vertices must not clash with themselves:

$$\text{edge\_list\_well\_formed } (v, \text{edges}) \iff$$

$$v \notin \text{set edges} \wedge \text{duplicate\_free edges}$$

- Graphs must not contain duplicate vertices:

$$\text{graph\_duplicate\_free []} \iff \text{T}$$

$$\text{graph\_duplicate\_free } ((r, rs) :: cs) \iff$$

$$(\forall rs'. (r, rs') \notin \text{set cs}) \wedge \text{graph\_duplicate\_free cs}$$

- Graphs must be symmetric – if  $v_1$  appears in the conflicts for  $v_2$ ,  $v_2$  appears in the conflicts for  $v_1$ :

`graph_reflects_conflicts cs`  $\iff$

$\forall r_1\ r_2\ rs_1\ rs_2.$

$\text{MEM}(r_1, rs_1)\ cs \wedge \text{MEM}(r_2, rs_2)\ cs \wedge \text{MEM}\ r_1\ rs_2 \Rightarrow \text{MEM}\ r_2\ rs_1$

These were all proven to hold of the graphs generated by the clash graph step

# Verified colouring algorithms

The first colouring algorithm verified was a naive one which simply assigns a new colour to each vertex:

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

naive_colouring constraints = naive_colouring_aux constraints 0
```

Correctness of naive\_colouring\_aux:

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

This implies the overall algorithm is correct:

$$\vdash (\forall n. \text{colouring\_satisfactory (naive\_colouring\_aux } cs \ n) \ cs) \Rightarrow \\ \text{colouring\_satisfactory (naive\_colouring } cs) \ cs$$

The naive algorithm isn't at all efficient. A better algorithm is the following, which assigns to each vertex the lowest colour which won't clash with its neighbours:

```
lowest_first_colouring [] = ( $\lambda x$ . 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)
```

This was also proved correct with respect to colouring\_satisfactory:

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

# Heuristics

More efficient colourings can be achieved by considering vertices in a different order

Heuristics re-order vertices based on some property – modelled as a sorting step before passing the graph to the colouring algorithm

A correct heuristic preserves the graph passed in. This means the resulting graph contains the same set of vertices and conflicts:

$\text{heuristic\_application\_ok } f \iff \forall \text{ list. set } (f \text{ list}) = \text{set list}$

Many heuristics are just sorts based on some property:

- Highest degree first
- Most uses first
- Longest live range first

# Smallest last

## A more complex heuristic

Remove the lowest-degree vertex from the graph, place it on a stack and repeat

Once the graph is empty, pop vertices off the stack and colour each one with the lowest available colour

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

smallest_last_heuristic cs =
smallest_last_heuristic_aux (λx. F)
  (sort_not_considered_by_degree (λx. F) cs) []

⊢ heuristic_application_ok smallest_last_heuristic
```



# Summary of correctness proof

- LVA returns exactly the variables which affect subsequent program behaviour
- Generated clash graphs contain exactly these conflicts and satisfy requirements for colouring algorithms
- Colouring algorithms generate colourings which are satisfactory with respect to the original graphs
- Colourings which are satisfactory on generated graphs are also fine with respect to the original definition of colouring correctness
- Colourings satisfying that definition generate code with the same execution behaviour

# Extension work

## Preference graphs

Preference graphs allow elimination of move instructions by placing source and destination in same register

Code was extended to include move instructions, and a function was added to map registers to lists of preferences

New colouring algorithm picks a preferred register where possible, and the lowest available otherwise

Verification was very similar to verification of the lowest-first algorithm

# Finite registers and spilling

No effect on colouring algorithms or proofs

Registers are spilled after allocation if they are out of range, and load/store instructions are inserted where necessary

This spill step was proven to preserve behaviour where memory is modelled as a second store

A most-uses-first heuristic was implemented to ensure frequently-used registers are prioritised, and this was proved correct:

$$\vdash \text{most\_used\_last\_heuristic } \textit{uses } \textit{list} = \\ \text{QSORT } (\lambda x y. \textit{uses } x < \textit{uses } y) \textit{ list}$$

(This puts frequently-used registers last because colouring algorithms work backwards from the end of the list)

# Conclusion

- Successful end-to-end verification of a register allocator
- Proofs are designed in a modular way so new algorithms and heuristics can be substituted in easily
- Future work:
  - Improved code representation
  - Performance of algorithms
  - More thorough treatment of register spilling