Developing a formally verified algorithm for register allocation

A Part III project

David Barker

9th June 2014

Introduction

The problem of register allocation

- Intermediate code assumes infinite registers
- Real machines have finite registers
- Using memory costs many cycles

Register allocation by graph colouring

Computing live ranges

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

$\{R_2,\ R_3\}$

$\{R_1,\ R_2,\ R_3\}$

$\{R_1,\ R_3,\ R_4\}$

$\{R_1,\ R_5\}$

Building a clash graph

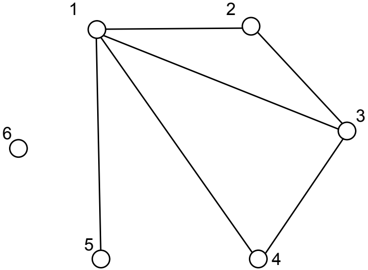
```
R1 = R2 + R3
R4 = R1 * R2
R5 = R3 - R4
R6 = R1 + R5
```

$\{R_2,\ R_3\}$

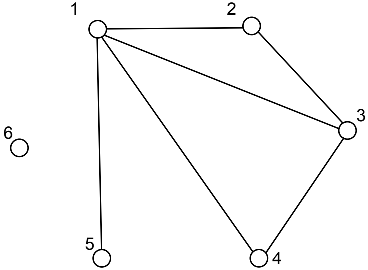
$\{R_1,\ R_2,\ R_3\}$

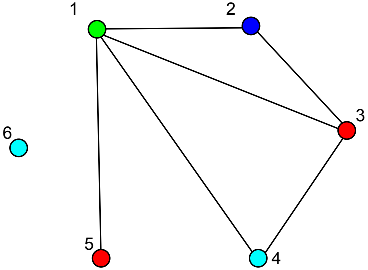
$\{R_1,\ R_3,\ R_4\}$

$\{R_1,\ R_5\}$

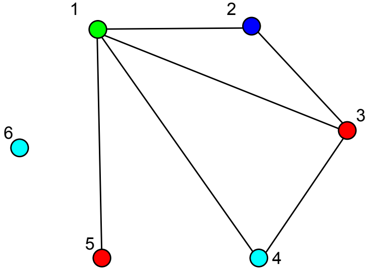


Colouring the clash graph



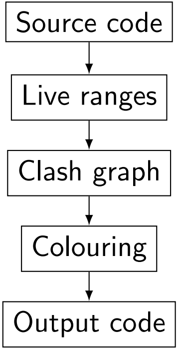


Applying the colouring




```
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:

$\texttt{inst = Inst } \textbf{of} \texttt{ num} \Rightarrow \texttt{num} \Rightarrow \texttt{num}$

This is evaluated on a store s as follows:

$\texttt{eval}\ f\ s\ \texttt{[]} = s$
$\texttt{eval}\ f\ s\ (\texttt{Inst}\ w\ r_1\ r_2 :: code) =$
$\texttt{eval}\ f\ ((w \texttt{ =+ } f\ (s\ r_1)\ (s\ r_2))\ s)\ code$

Colourings are functions of type $num \to num$

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:

$\texttt{insert}\ x\ xs\ \texttt{=}\ \textbf{if}\ \texttt{MEM}\ x\ xs\ \textbf{then}\ xs\ \textbf{else}\ x\texttt{::}xs$

$\texttt{delete}\ x\ xs\ \texttt{=}\ \texttt{FILTER}\ (\lambda y\texttt{.}\ x \neq y)\ xs$

The algorithm

Live variable analysis

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

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

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:

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

Clash graph generation

Clash graph representation

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

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

Here r_n is the n^{th} register and c_n is the n^{th} 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 (λset. 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 (λreg. (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`)

  $\vdash r \notin \texttt{set (conflicts\_for\_register } r\ code\ live\texttt{)}$

- The graph is complete: any registers from the same conflicting set appear in each other's conflicts

  $\vdash \texttt{MEM } c\ \texttt{(conflicting\_sets } code\ live\texttt{)} \wedge \texttt{MEM } r\ c \wedge \texttt{MEM } s\ c \wedge$
  $r \neq s \Rightarrow$
  $\texttt{MEM } r\ \texttt{(conflicts\_for\_register } s\ code\ live\texttt{)}$

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

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

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:

$\texttt{colouring\_satisfactory}\ \mathit{col}\ \texttt{[]}\ \iff\ \texttt{T}$
$\texttt{colouring\_satisfactory}\ \mathit{col}\ \texttt{((}r\texttt{,}\mathit{rs}\texttt{)::}\mathit{cs}\texttt{)}\ \iff$
$\mathit{col}\ r \notin \texttt{set (MAP}\ \mathit{col}\ \mathit{rs}\texttt{)} \wedge \texttt{colouring\_satisfactory}\ \mathit{col}\ \mathit{cs}$

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

$\vdash \texttt{duplicate\_free}\ \mathit{live} \Rightarrow$
$\quad \texttt{colouring\_satisfactory}\ c\ \texttt{(get\_conflicts}\ \mathit{code}\ \mathit{live}\texttt{)} \Rightarrow$
$\quad \texttt{colouring\_ok\_alt}\ c\ \mathit{code}\ \mathit{live}$

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:

  $\texttt{edge\_list\_well\_formed}\ (v, \mathit{edges}) \iff$
  $v \notin \texttt{set}\ \mathit{edges} \wedge \texttt{duplicate\_free}\ \mathit{edges}$

- Graphs must not contain duplicate vertices:

  $\texttt{graph\_duplicate\_free}\ [] \iff \texttt{T}$
  $\texttt{graph\_duplicate\_free}\ ((r, \mathit{rs})::\mathit{cs}) \iff$
  $(\forall \mathit{rs}'.\ (r, \mathit{rs}') \notin \texttt{set}\ \mathit{cs}) \wedge \texttt{graph\_duplicate\_free}\ \mathit{cs}$

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

  $$\begin{array}{l}\texttt{graph\_reflects\_conflicts}\ cs\ \iff \\ \forall\, r_1\ \ r_2\ \ rs_1\ \ rs_2\,. \\ \quad \texttt{MEM}\ (r_1, rs_1)\ cs\ \land\ \texttt{MEM}\ (r_2, rs_2)\ cs\ \land\ \texttt{MEM}\ r_1\ \ rs_2\ \Rightarrow\ \texttt{MEM}\ r_2\ \ rs_1\end{array}$$

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 = ($\lambda x.\ n$)
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:

⊢ graph_edge_lists_well_formed cs ⇒
$\forall n.$ colouring_satisfactory (naive_colouring_aux cs n) cs

This implies the overall algorithm is correct:

⊢ ($\forall n.$ colouring_satisfactory (naive_colouring_aux cs n) cs) ⇒
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 [] = (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)
```

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:

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

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) []
```

$\vdash$ 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:

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

(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