Developing a formally verified algorithm for register allocation

A Part III project

David Barker

9^{th} June 2014

The problem of register allocation

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

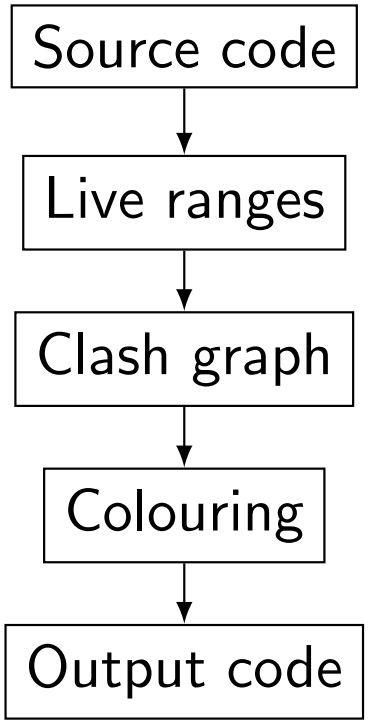
Computing live ranges

Building a clash graph

Colouring the clash graph

Applying the colouring

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 ⇒ num ⇒ 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

Simple set functions:

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

delete x xs = FILTER (λy. x ≠ y) xs

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

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

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:

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

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

The first of these states that registers never conflict with themselves, and follows easily from the definition of `conflicts_for_register`:

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

The second states that the graph is complete – any registers from the same conflicting set appear in each other's conflicts:

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

Finally, it was shown that the graph doesn't contain any false conflicts – every conflict is the result of two registers appearing in a conflicting set together:

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

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:

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

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

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

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 graphs passed satisfied several properties:

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

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

- Graphs must not contain duplicate vertices:

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

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

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

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
```

The statement of `naive_colouring_aux`'s correctness is given below:

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

It was then shown that this implies the overall algorithm is correct:

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

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 proven correct with respect to `colouring_satisfactory`:

$$\vdash \texttt{graph\_reflects\_conflicts}\ cs \wedge \texttt{graph\_duplicate\_free}\ cs \wedge \texttt{graph\_edge\_lists\_well\_formed}\ cs \Rightarrow \texttt{colouring\_satisfactory}\ (\texttt{lowest\_first\_colouring}\ cs)\ cs$$