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

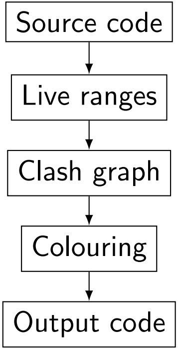
Computing live ranges

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 $\Rightarrow$ num $\Rightarrow$ num

This is evaluated on a store s as follows:

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

Colourings are functions of type $num \rightarrow num$

Colourings can be applied simply by substituting registers:

$\texttt{apply}\ c\ \texttt{[] = []}$
$\texttt{apply}\ c\ \texttt{(Inst}\ w\ r_1\ r_2\texttt{::}code\texttt{) =}$
$\texttt{Inst (}c\ w\texttt{) (}c\ r_1\texttt{) (}c\ r_2\texttt{)::apply}\ c\ code$

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:

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

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

$\texttt{list\_union}\ \texttt{[]}\ ys\ \texttt{=}\ ys$
$\texttt{list\_union}\ (x\texttt{::}xs)\ ys\ \texttt{=}\ \texttt{insert}\ x\ (\texttt{list\_union}\ xs\ ys)$

$\texttt{list\_union\_flatten}\ \texttt{[]}\ \texttt{=}\ \texttt{[]}$
$\texttt{list\_union\_flatten}\ (l\texttt{::}ls)\ \texttt{=}$
$\texttt{list\_union}\ l\ (\texttt{list\_union\_flatten}\ ls)$

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:

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