Register Allocation

Objectives

May need to insert loads and stores to move values between registers and stack.

Goal: to make effective use of the target machine's register set

Goal: to minimize the number of loads and stores

Approaches

We look at three approaches, not necessarily exclusive:

- Liveness and interference
- Usage counts
- Furthest invocation

Liveness Analysis

A variable is *live* if the variable is used at some later point in the program and there is not an intervening assignment to the variable.

Example:

```
1 (movq (int 5) (var a))
2 (movq (int 30) (var b))
3 (movq (var a) (var c))
4 (movq (int 10) (var b))
5 (addq (var b) (var c))
```

Are `a` and `b` live at the same time?

Calculating Liveness

Traverse the instruction sequence back to front (i.e., backwards in execution order).

Let $I_1, \ldots, I_n$ be an instruction sequence.

We write:

- $L_{after}(k)$ for the set of live variables after instruction I_k
- $L_{before}(k)$ for the set of live variables before instruction I_k

The live variables after an instruction are always the same as those before the next instruction:

- $L_{after}(k) = L_{before}(k+1)$

Furthermore:

- $L_{after}(n) = \{\}$

Calculating Liveness

At each instruction, we calculate:

- $L_{before}(k) = (L_{after}(k) - W(k)) \cup R(k)$

Where $W(k)$ are the variables **written** by instruction I_k

And $R(k)$ are the variables **read** by instruction I_k

Calculating Liveness

Consider the following example:

Source program:

```
(program
  (let ([v 1])
  (let ([w 46])
  (let ([x (+ v 7)])
  (let ([y (+ 4 x)])
  (let ([z (+ x w)])
      (+ z (- y))))))))
```

1	(program(v w x y z t.1 t.2)	
		{}
2	(movq (int 1) (var v))	
		{v}
3	(movq (int 46) (var w))	
		{v, w}
4	(movq (var v) (var x))	
		{w, x}
5	(addq (int 7) (var x))	
		{w, x}
6	(movq (var x) (var y))	
		{w, x, y}
7	(addq (int 4) (var y))	
		{w, x, y}
8	(movq (var x) (var z))	
		{w, y, z}
9	(addq (var w) (var z))	
		{y, z}
10	(movq (var y) (var t.1))	
		{t.1, z}
11	(negq (var t.1))	
		{t.1, z}
12	(movq (var z) (var t.2))	
		{t.1, t.2}
13	(addq (var t.1) (var t.2))	
		{t.2}
14	(movq (var t.2) (reg rax)))	
		{}

Interference Graphs

Based on the liveness analysis, we know where each variable is needed.

During register allocation, we need to answer questions of the specific form: are variables *u* and *v* live at the same time?

In that case, *u* and *v* cannot be assigned to the same register.

An *interference graph* is an undirected graph that has an edge between two variables if they are live at the same time, that is, if they interfere with each other.

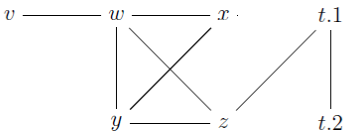
Calculating Interference Graphs

If instruction I_k is:

- A move: `(movq s d)`, then add the edge (d, v) for every $v \in L_{\texttt{after}}(k)$ unless $v = d$ or $v = s$.
- **Not** a move but some other instruction such as `(add s d)`, then add the edge (d, v) for every $v \in L_{\texttt{after}}(k)$ unless $v = d$.
- Of the form `(callq label)`, then add an edge (r, v) for every caller-save register r and every variable $v \in L_{\texttt{after}}(k)$.

Interference Graph for our Example

1	(program (v w x y z t.1 t.2)	
		{}
2	(movq (int 1) (var v))	
		{v}
3	(movq (int 46) (var w))	
		{v, w}
4	(movq (var v) (var x))	
		{w, x}
5	(addq (int 7) (var x))	
		{w, x}
6	(movq (var x) (var y))	
		{w, x, y}
7	(addq (int 4) (var y))	
		{w, x, y}
8	(movq (var x) (var z))	
		{w, y, z}
9	(addq (var w) (var z))	
		{y, z}
10	(movq (var y) (var t.1))	
		{t.1, z}
11	(negq (var t.1))	
		{t.1, z}
12	(movq (var z) (var t.2))	
		{t.1, t.2}
13	(addq (var t.1) (var t.2))	
		{t.2}
14	(movq (var t.2) (reg rax)))	
		{}



Still a bit short on Registers?

Spill a register on the stack.

Use the one for which the next use is furthest in the future.

- Return index in the block of the next reference to a register.
- Can be precomputed in a backward pass over the block.