Linear Scan Register Compression

Philip Blair

18 September 2016

This paper describes the variant of Poletto and Sarkar[1]'s register allocation algorithm that Pyret uses in order to compress the number of JavaScript variables included in compiled programs' output.

Figure 1 shows the algorithm used. It differs from the algorithm in [1] primarily in how spilling is handled. To explain why, let us first have some context:

While we are trying to minimize the number of 'registers' used by the compiled output, we are not restricted to any specific number. This gives us some interesting differences from traditional register allocation:

1. **Uniform Cost:** Once a location has been 'allocated', it costs exactly the same to use as any other already-allocated location (that is, nothing is assigned to a location with a relative cost similar to spilling over from registers into memory in traditional register allocation).

2. **The "Hilbert Register" Effect:** If all registers are in use at any given time, there is always another register available.

As such, our algorithm's spilling handling is much more streamlined. For one, there is no need to examine the lastmost-ending interval allocated in the register pool[1], for there is no extra "cost" associated with leaving it where it is. Additionally, instead of placing the spilled variable on the stack, we simply can pretend as though there was an additional register (namely, itself) which was available in the register pool.

References

[1] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 9 1999.

[1] This is done in SpillAtInterval in [1]

```
(1)  begin
(3)    I ⊆ V × ℕ × ℕ;          (Set of intervals from liveness analysis [input])
(4)    comment: Entries in I are of form (variable,start loc.,end loc.)
(5)    P := ∅;                                   (Pool of available registers)
(6)    R ⊆ V × V := ∅;                           (Allocations dictionary)
(7)    A ⊆ I := ∅;                               (Active intervals)
(8)    foreach i in I do
(9)      prune-dead(i);
(10)     A := A ∪ {i};
(11)                                             (Mark i as active)
(12)     (v_cur, s_cur, e_cur) := i;
(13)     if |P| > 0
(14)       then let r ∈ P;
(15)            R[v_cur] := r;                   (Allocate r for v_cur)
(16)            P := P \ {r};                    (Remove r from avail. pool)
(17)        else R[v_cur] := v_cur;              (Grow pool to accomodate)
(18)     fi
(19)   od
(20) where
(21) proc prune-dead(i) ≡
(22)   (v_cur, s_cur, e_cur) := i;
(23)   if |A| > 0
(24)     then (v_min, s_min, e_min) := peek-min(A); (int. with minimum e_min in A)
(25)          if e_min < s_cur
(26)            then delete-min(A);
(27)                 r := R[v_min];
(28)                 P := P ∪ {r_min};
(29)                 comment: There may be other intervals to prune, so recur.
(30)                 prune-dead(i);
(31)          fi
(32)   fi.
(33) end
```

Figure 1: Register Compression Algorithm Used in Pyret