SETL Newsletter #24 April 20, 1971

Description of a Register Allocation Algorithm K. Kennedy

This will describe an algorithm due to Horowitz, Karp, Miller, and Winograd [1] for allocating index registers for straight line code. Since we are only interested in the indices referenced at each step, we use the term program to mean a sequence of indices, one for each step. At any step the index referenced may be modified. Such a modifying reference is indicated by an asterisk following the index.

In SETL we will represent the program as a sequence of ordered pairs: the first element of the ordered pair at step i will be the index referenced; the second element will be a flag which will be true if the index is modified at that step and false otherwise. The following example shows a program and its corresponding SETL sequence.

program	SETL sequence
x_1	$\langle 1, \langle x_1, \underline{f} \rangle \rangle$
x_2	$\langle 2, \langle x_2, \underline{f} \rangle \rangle$
x_3^*	$\langle 3, \langle x_3, \underline{t} \rangle \rangle$
x_2^*	$\langle 4, \langle x_2, \underline{t} \rangle \rangle$
x_3	$\langle 5, \langle x_3, \underline{f} \rangle \rangle$
x_1	$\langle 6, \langle x_1, \underline{f} \rangle \rangle$
x_2	$\langle 7, \langle x_2, \underline{f} \rangle \rangle$

Now suppose we have a machine with N index registers. A register allocation regs for the program is a sequence of register configurations, one for each step. A register configuration for step i is a set of N ordered pairs which represent the contents of the registers at that step. Each ordered pair is of the form <index, flag> where flag is t if the index is modified (i.e. it must be stored if it is to be replaced in registers). If the allocation is to be a legal one, the index called for at step i (for all i) must be in registers at that step. In other words

(1 ≤ ∀i ≤ #program) program (i) ε regs{i};

Associated with each allocation is a cost which is determined by the following formula: suppose we can define a function cost(config1,config2) which represents the cost of going from one configuration to another. Then the cost associated with an allocation regs is:

$$\text{total cost} = \sum_{i=2}^{\#\text{program}} \text{cost}(\text{regs}\{i-1\},\text{regs}\{i\})$$

Note that regs is a set of ordered triples <step,index,flag> so the configuration at step i (regs{i}) is a set of ordered pairs <index,flag>. The cost function may be defined as follows:

1) the cost of replacing a modified index in a configuration is 2 (1 store, 1 load).
2) the cost of replacing an unmodified index in a configuration is 1 (1 load).
3) the cost of replacing a modified index with the same index unmodified is 1 (1 store).
4) the cost of replacing an unmodified index with a modified version of the same index is 0.

Using these rules we may define the function cost in SETL as follows:

```
definef cost(con1,con2); c=0;
  (∀x ε hd[con1]) if n x ε hd[con2]
      then c = c + if con1(x) then 2 else 1;
      else c = c + if con1(x) and n con2(x) then 1 else 0;
        end if; end ∀x; return c; end cost;
```

A possible register allocation algorithm is to generate all possible legal configurations at each step and then pick the allocation (one configuration from each step) with minimal cost. Since there are a finite number of such allocation , the algorithm will terminate.

However, this is a bit too combinatorial to be practical so Horowitz et al. show how the number of allocations to be considered may be significantly reduced.

The basic idea of the Horowitz algorithm is to start with the initial configuration and generate for step 1 a restricted class of configurations reachable from the initial configuration by minimal change branches. Once one has the configurations associated with step i one can generate all configurations for step i+1 which can be reached from these configurations by minimal change branches. This process can continue until we reach the last step. If we associate with each configuration at step i a weight (defined as the minimal cost of reaching this configuration at this step) and a function parent(con) which maps a configuration at step i onto a configuration at step i-1 s.t.

weight(con) = weight(parent(con)) + cost(parent(con),con) ,

we can find the best allocation by finding the configuration associated with the last step which has minimal weight, and then following the chain of pointers back to the start. In other words, if conmin is the minimal configuration at the last step then

```
regs{#program} = conmin
regs{#program - 1} = parent(conmin)
                etc.
```

In fact our data structures will be slightly more complex. Associated with each step we will have a set of nodes (which are atoms). These nodes are mapped onto corresponding configurations by the function config. The function parent will map the node onto the node from which it can be reached by a minimal change branch s.t.

weight(node) = weight(parent(node))+cost(config(parent(node)), config(node))

weight is the function which defines the weight of a node.

These three functions are the main data structures involved in the algorithm. The branch from node n(stepi-1) to node n' (step i) is a minimal change branch if one of the following conditions holds:

1) the configuration for n is identical to the configuration for n'
2) the index called for at step i is not in the configuration for n and config(n') differs from config(n) in exactly one element -- the index referenced at step i replaces some index in config(n).
3) the index called for by step i is modified at that step and appears unmodified in config(n). Then config(n') differs from config(n) only in that an unmodified occurrence of that index appears in config(n) where a modified occurrence appears in n'.

The overall allocation process looks like this in SETL:

```
define allocate(program,regs,ic); nodes = {ic}
 weight(ic) = 0; parent = nl; weight = nl
 newer = nl; newnodes = nodes;
 (1 ≤ ∀step ≤ #program) cleanse newnodes;
   (∀ n ε newnodes) generate n; end ∀n;
/*generate will put the nodes for step in the set newer*/
 newnodes = newer;  newer = nl;  end ∀step;
 findalloc newnodes; end allocate;
```

The routine generate generates nodes for step n from nodes for step -1. As we shall see, it generates a restricted class of those nodes which can be reached from n by minimal change branches. The improvements will be described later. The routine findalloc picks the best allocation by finding the minimal weight node associated

with the last step and working back via parent.

In SETL this goes as follows:

```
define findalloc newnodes; allocate external
  regs, weight, parent, config, program;
  minnode = tl[minhd: n ε newnodes]<weight(n),n>;
  k = #program; (while k ge 1 doing k=k-1; minnode=parent(minnode);)
          regs{k}=config(minnode); end while;
      end findalloc;
```

The function minhd is defined

```
definef xminhd y;
    return if hd y lt hd x then y else x;
    end minhd;
```

The function maxhd to be used later is defined similarly.

The routine cleanse is a routine which eliminates generated nodes according to the rule:

> If n and n' are two nodes associated with step i such that weight(n) + cost(config(n),config(n')) ≤ weight(n') we may eliminate n'.

This rule may be coded in SETL as follows:

```
define cleanse nodeset; allocate external weight, cost,
      parent, config;
(∀x ε nodeset) if (∃ y ε (nodeset less x) |
  weight(y) + cost(config(y),config(x)) le weight(x))
  then parent(x) = Ω; weight(x) = Ω; config(x) = Ω;
  nodeset = nodeset less x; end if; end ∀x; end cleanse;
```

The subroutine generate will require a certain amount of discussion. Suppose n is a node associated with step i-1. The simplest possible routine would generate all nodes for step i which could be reached from n by a minimal change branch. Thus, if x (or x*) is referenced at step i then

1. If x is in config(n) a single node n' is generated such that config(n') is identical to config(n) except that the flag for x might be true in n' but false in n.
2. If x is not in config(n) then N new nodes $(n_1,\ldots,n_N)$ are generated where n_i is formed from n by replacing the i-th component of config(n) by x.

This is not quite good enough, however, because we would like to reduce the number of nodes generated by case 2. We will do this by using lookahead.

First, we must define two lookahead functions:

1. next(i,x) will return the number of steps to the next use (modifying or non-modifying) of x.

In SETL:

```
definef next(i,x); allocate external program;
 return if(i <∃[k] ≤ #program | hd program(k) eq x)
        then k-i else #program+1-i,
        end next;
```

2. next(i,x) will return the number of steps to the next modifying use of x after step i.

In SETL:

```
definef nexts(i,x); allocate external program;
 return if(i <∃[k] ≤ #program | program(k) eq<x,t>)
        then k-i; else #program+1-i;  end nexts;
```

Using these two functions we may limit the number of nodes that we must generate at each step. Suppose, once again, that step i of our program calls for x and that node n (associated with step i-1) does not have x or x* in its configuration. The following observations are due to Horowitz et al.

1. If there are several unmodified variables in config(n) we need never generate more than one node in which x replaces an unmodified variable. The unmodified variable to be replaced can always be chosen to be the variable y ∈ hd[config(n)] s.t. next(i,y) is a maximum. There may be more than one unmodified variable, say y and z, s.t. next(i,y) = next(i,z) = #program + 1 ; in this case we can still arbitrarily pick the one to be replaced (either y or z will do).

2. Suppose M = next(i,y) where y is the unmodified variable which is used farthest away (picked above). We need never generate a node in which x replaces a modified variable z if next(i,z) ≤ M. (Actually this is an improvement over the condition nexts(i,z) ≤ M stated by Horowitz et al.)

3. Suppose K = next(i,z) where z is the modified variable in config(n) which is first used farthest away. We need not generate a node in which x replaces w if w is modified and nexts(i,w) < K. If K = program + 1 then we need only generate a single node in which x replaces z.

The algorithm given below works with the constants M and K by finding two ordered pairs:

maxu = <M,y> ; maxs = <K,z>

It will then generate a node in which x replaces y; if K = #program it will generate a node in which x replaces z; if #program+1 > K > M it will generate nodes in which x replaces any modified variable w in config(n) such that nexts(i,w) ≥ K.

In SETL the algorithm is a long one and goes as follows:

```
define generate n; /*n is a node generated at the last
  step from which nodes are to be generated for the
  current step */
allocate external newnodes, newer, program, step, parent,
  weight, config;  con = config(n);
/* see if program(step) is in registers */
  if ∃[x] ε con | hd x eq hd program(step) then do genode(x);
  if tl x then config(nl) = config(nl) less prog(step) with x;
  end if;
  weight(nl)=weight(n);       return; end if ∃;
block genode(x); nl=newat; nl in newer; parent(nl)=n
  config(nl) = config(n) less x with program(step); end genode;
/* else look for unmodified and modified indices whose
  next uses are farthest away */
  maxu = [maxhd: x ε con | n tl x]<next(step,hd x),x>
  maxs = [maxhd: x ε con | tl x]<next(step,hd x),x>
/* generate node replacing most distant unmodified index */
  if maxu ne Ω then do genode (tl maxu);
   weight(nl) = weight(n)+1; if maxs eq Ω then return;;
     if hd maxu ge hd maxs then return;;
else /* define maxu if undefined */
  maxu = <0,nl>; end if maxu;
/* now generate nodes replacing modified indices */
  if hd maxs eq (#program+1) then do genode (tl maxs); return;;
/*else*/ (∀ x ε con | tl x and next(step,hd x) gt hd maxu
        and nexts(step,hd x) ge hd maxs]
  do genode(x); weight(nl)=weight(n)+2; end ∀x
    end generate;
```

This allocation scheme should generate a minimal number of nodes and operate very fast. Actual tests have shown that the algorithm is very practical and useful.