SETL Newsletter #102 K. Kennedy

Reduction in Strength Using Hashed Temporaries March 12, 1973

This newsletter describes a simple 'reduction in strength' algorithm which works on strongly-connected regions.

Intermediate Code

Since reduction in strength is one of a class of algorithms which must examine code in some form, we must begin by describing an intermediate code form. The intermediate code we choose is simple, yet flexible enough to support the optimization methods we propose to describe.

We take the intermediate code with which we will be concerned to be a set of SETL blank atoms (formed by successive calls to the primitive newat) with which several mappings are associated. Let *at* be an atom of code.

1. op(at) is the operation code for the instruction.
   The following operations are available:

Operation	Code	#Arguments
nop	0	0
add	1	2
sub	2	2
mul	3	2
div	4	2
exp	5	2
xld (indexed load)	6	2
sto	7	1
neg (store negative)	8	1
xst (indexed store)	9	2
br (branch)	10	0
brc (branch conditionally)	11	2
bsr (branch to subroutine)	12	variable
bfn (branch to function)	13	variable
hlt	14	0

The operations 0-5 are reasonably self-explanatory.
"Indexed load" (xld) takes an array name a and a simple variable i as its arguments and loads the value of $a(i)$ into the target variable.

"Store" (sto) moves the value of its argument to the target. "Store negative" (neg) negates the value of its argument before moving it to the target. "Indexed store" (xst) has an array name *a* as its target and an index *i* and a simple variable *x* as its arguents. Its effect is to move the value of *x* to *a(i)*. The branch instructions use a flow structure that will be described later. "Branch conditionally" (brc) accepts two arguments, x1 and x2, and causes a branch if x1 $\geq$ x2. The other branch instructions are self-explanatory except for the fact that br and bsr have no targets while bfn has a target which receives the value of the function. In the algorithms which follow we assume that the mnemonics listed above are pseudonyms for the opcodes specified.

2. targ(at) is the name of the target variable for the instruction if a target exists.
3. args(at) is a tuple containing the names of the arguments to the instruction.
4. next(at) is the next instruction to be taken. Its value is a single atom for most instructions. However, for the branch instructions its value is a pair

   <at1,at2>

   where *at1* is the next instruction in code sequence and *at2* is the branch target. In the case of a branch to a subroutine, the branch target may be a name instead of an atom.

In manipulating two-argument instructions we use the following SETL macros:

```
macro arg1(at); hd args(at) endm arg1;

macro arg2(at); args(at)(2) endm arg2;
```

which allow us to access the first and second arguments.

Our algorithms will often insert instructions into the code contained within a strongly-connected program region, so we will need a subroutine to insert an instruction after another instruction. We will never insert code immediately after a branch instruction so the complicated flow problems which such insertion would imply need not be dealt with.

The routine *insert* presented below has 5 arguments.

1. at - the instruction after which the new instruction is to be inserted
2. t - the target of the new instruction
3. o - the opcode for the new instruction
4. a - the argument tuple for the new instruction
5. c - the code set into which the new instruction is to be inserted.

Here is the SETL code.

```
define insert(at,t,o,a,c);
/*  get new blank atom  */
node = newat;
/* initialize functions */
<targ(node), op(node), args(node)> = <t,o,a>;
/* set up flow functions */
<next(node), next(at)> = <next(at), node>;
/* add node to code set */
c = c with node;
return;
end insert;
```

2. Finding Induction Variables.

One of the first subtasks of the reduction in strength process is to locate the induction variables appearing in a strongly-connected program region. In general, induction variables are those variables which are defined in terms of region constants and other induction variables by operations of the following form:

$$x \leftarrow \pm y$$

$$x \leftarrow y \pm z$$

where y and z are either region constants or induction variables. To locate induction variables, one must use a process of elimination. We therefore propose the following scheme for finding variables which are not induction variables. Let IV be the set of all induction variables and RC the set of region constants for the strongly-connected region.

1. If x ← op(y,z) and op is not one of {neg,add,sub,sto} then x is not in IV.
2. If ∃ an instruction s in the strongly-connected region such that op(s) ∈ {bsr,bfn} and x ∈ args(s) then x is not an induction variable. This restriction eliminates the possibility that a subroutine side effect shall modify an induction variable.
3. If x ← op(y,z) and y or z is not an element of IV ∪ RC then x is not in IV.

The algorithm for finding induction variables proceeds by passing through the nodes of the set *scr* (the strongly-connected region) and collecting all variables which are targets of {add,sub,sto,neg} into a set *iv* while collecting all subroutine arguments into a set *subargs*. The difference between these two sets gives the initial approximation to the set of induction variables. We then pass through the code repeatedly applying restriction 3 until no more variables are eliminated from *iv*.
The routine *findivars* is coded in SETL as follows.

```
definef findivars(scr,rc);
/* scr is the set of nodes in the strongly-connected region,
   rc  is the set of region constants,
   iv  is the set of induction variables,
   subargs is the set of variables which are arguments to subroutines,
   ivnodes is the set of instruction nodes which set  variables in iv*/
<iv,subargs,ivnodes> = <nℓ,nℓ,nℓ>;
```

```
/* pass through the region applying rules 1 and 2 to get
   the initial approximation for iv */
(∀n ∈ scr)
  if op(n) ∈ {add,sub,sto,neg}
  then /* rule 1 */
    iv = iv with targ(n);
    ivnodes = ivnodes with n;
  else if op(n) ∈ {bsr,bfn}
  then /* rule 2 */
    subargs = subargs + {(args(n))(i), 1≤i≤#args(n)};
  end if;
end ∀n;
/* take the difference to form the approximation */
iv = iv - subargs;
ivnodes = {n ∈ ivnodes | targ(n) n ∈ subargs};
/* we can now restrict our attention to ivnodes --the set of
   instructions which set possible induction variables. we pass
   through iv nodes eliminating induction variables which do not
   obey restriction 3 */
oldiv = nℓ;
(while iv ne oldiv) oldiv = iv;
  (∀n ∈ ivnodes|arg1(n) n ∈ (iv+rc)
    or arg2(n) n ∈ (iv+rc))
  iv = iv less targ(n);
  end ∀n;
/* reduce ivnodes */
  ivnodes = {n ∈ ivnodes|targ(n) ∈ iv}
end while;
return <iv,ivnodes>;
end findivars;
```

Note that the value returned by the function *findivars* is a pair, consisting of the set of induction variables and the set of instructions which set those variables.

3. Finding Candidates for Reduction.

The algorithm we will present will aim to reduce all multiplications of the form

$$i * c$$

where i is an induction variable and c is a region constant. These can be found by passing through the region and checking the arguments of multiplications. The following routine *findcands* returns the set of nodes which represent operations of the appropriate form.

```
definef findcands(scr,rc,iv)
/* scr is the region,   rc is the set of region constants,
   iv  is the set of induction variables */
/* initialize */
cands = nℓ;
/* pass through scr looking at multiplications */
(∀at ∈ scr | op(at) eq mul)
   if arg1(at) ∈ iv and
      arg2(at) ∈ rc
   then cands = cands with at;
   else if arg2(at) ∈ iv and
           arg1(at) ∈ rc
   then /* switch arguments to establish canonical form */
      <arg1(at),arg2(at)> = <arg2(at),arg1(at)>;
       cands = cands with at;
   end if;
end ∀at;
return cands;
end findcands;
```

4. The Temporary Table.

The idea of reduction in strength is to replace

$$x \leftarrow i * c$$

by

$$x \leftarrow t$$

where t is a temporary which holds the current value of i * c over the entire region. In the present package of algorithms these temporaries will be accessed through a hash table which uses the names of the operands of the multiplication as keys. Thus

$$t_{i*c}$$

will contain the value of i*c in the region. In using this trick we must do two things to assure that t_{i*c} always contains the correct value.

1) An initilization of the form $t_{i*c} \leftarrow i*c$ must be inserted just prior to entry to the scr. For this purpose, it is useful to assume that each strongly-connected region has a *prolog* -- a basic block which is always executed just prior to entering the region. New initializations will be inserted at the end of the prolog.
2) After each instruction which sets i we must insert an instruction which modifies the vlaue of t_{i*c} appropriately. This is not as simple as it sounds since instructions of the following forms can occur.

Instruction	Operation to be Inserted
$i \leftarrow c_2$	$t_{i*c} \leftarrow t_{c_2*c}$
$i \leftarrow -c_2$	$t_{i*c} \leftarrow -t_{c_2*c}$
$i \leftarrow j + c_2$	$t_{i*c} \leftarrow t_{j*c} + t_{c_2*c}$
$i \leftarrow j - c_2$	$t_{i*c} \leftarrow t_{j*c} - t_{c_2*c}$

This table shows that we must not only create temporaries for i*c but also for j*c and c_2*c for every j and c_2 that can affect the value of i. In addition, we must insert initializations and modifications for these temporaries. Thus we must find all induction variables and region constants.that can affect the value of i. To handle all the cases which can arise we develop a routine which computes a set *affect* of ordered pairs

<i,x>

where *i* is an induction variable and *x* is an induction variable or region constant which can affect it.

The idea is to pass through the scr building an initial *affect* relation from instructions which set induction variables and then to fill in all necessary addition items using a process of transitive closure. The following routine returns the set *affect*.

```
definef findaffect(ivnodes,iv,rc);
/* ivnodes is the set of nodes which set induction variables,
   iv is the set of induction variables, rc is the set of region
   constants */
/* initialize so that each iv affects itself */
affect = {<x,x>, x ∈ iv};
/* pass through ivnodes to get the initial relation -- any
   operands of an instruction which sets x must affect x */
(∀at ∈ ivnodes) x = targ(at);
   if pair args(at) then
      affect{x} = affect{x} + {<x,arg1(at)>,<x,arg2(at)>};
   else affect{x} = affect{x} with <x,arg1(at)>;
   end if;
end ∀at;
/* now take the transitive closure by adding to affect{x} any
   variable which affects an induction variable in affect{x} */
n = 0;
(while #affect gt n) n = #affect;
   (∀x ∈ iv) affect{x} = affect{x}
      +{< x,y>, y ∈ affect[iv * affect{x}]};
   end ∀x;
end while;
return affect;
end findaffect;
```

Once we have the *affect* set we can accomplish strength reduction very neatly using the temporary table. If i*c is a candidate for reduction, we must form

$$t_{x*c} \text{ for all } x \in \text{affect}\{i\}$$

inserting appropriate initializations and modifications for these temporaries.

In the setl routine which follows we assume a mapping t such that t(x,y) maps x and y to the unique compiler-generated name for t_{x*y}. (In standard practice, this mapping would be realized by a hashed table.) The initialization instruction for each temporary will be inserted at the end of the prolog when the entry for that temporary is inserted in the table. This will require a pointer *plast* to the last instruction in the prolog.

The algorithm presented below takes the candidates one at a time and performs reduction for them. Note that in implementing such a routine, efficiency could be improved substantially by using more parallelism.

```
define    streduce(prolog,plast,scr,rc);
  /* prolog is the initialization block whose last instruction is
     plast, scr is the region and rc are the region constants which
     we assume are found in an earlier code-motion pass */
  /* find induction variables */
  <iv,ivnodes> = findivars(scr,rc);
  /* find candidates for reduction */
  cands = findcands(scr,rc,iv);
  /* find the affect relation */
  affect = findaffect(ivnodes,iv,rc);
  /* now pass through the candidates creating temporaries and
     inserting initializations and modifications */
  (∀at ∈ cands) x = arg1(at);  c = arg2(at);
  /* create the new temporaries as required */
     (∀y∈affect{x}| t(y,c) = Ω)
        t(y,c)=newtemp; /*compiler generated name */
  /* initialization in prolog */
```

```
        insert(plast,t(y,c),mul,<y,c>,prolog);
        plast = next(plast);
     /* double entries for const * const */
        if y ∈ rc then t(c,y) = t(y,c);;
     /* insert modifications to the new temporaries after
        instructions which set induction variables */
        (∀n ∈ ivnodes | targ(n) eq y)
           newargs = if pair args(n)
              then <t(arg1(n),c),t(arg2(n),c)>
              else <t(arg1(n),c)>;
     /*the inserted instruction has the target t(y,c), the same
        operations as n, and newargs as its argument */
           insert(n,t(y,c), op(n), newargs, scr);
        end ∀n;
     end ∀y;
     /* now replace the candidate by a store operation */
     <op(at),args(at)> = <sto,<t(x,c)>>;
  end ∀at;
end streduce;
```

This completes the presentation of our strength reduction algorithm. An example will show what this algorithm will do.

Original Code:

```
prolog {  i = 1
          j = 1
              :
       {  i = j+1
              :
          x = j*5
              :
region {  j = i+3
              :
          y = i*6
              :
          j = j+1
```

After reduction:

prolog

$$
\begin{aligned}
i &= 1 \\
j &= 1 \\
t_{i*5} &= i*5 \\
t_{j*5} &= j*5 \\
t_{j*6} &= j*6 \\
t_{i*6} &= i*6 \\
t_{1*5} &= 5 \\
t_{1*6} &= 6 \\
t_{3*5} &= 15 \\
t_{3*6} &= 18
\end{aligned}
$$

region

$$
\begin{aligned}
&\vdots \\
j &= j + 1 \\
t_{i*5} &= t_{j*5} + t_{1*5} \\
t_{i*6} &= t_{j*6} + t_{1*6} \\
&\vdots \\
x &= t_{j*5} \\
&\vdots \\
j &= i + 3 \\
t_{j*5} &= t_{i*5} + t_{3*5} \\
t_{j*6} &= t_{i*6} + t_{3*6} \\
&\vdots \\
y &= t_{i*6} \\
&\vdots \\
j &= j + 1 \\
t_{j*5} &= t_{j*5} + t_{1*5} \\
t_{j*6} &= t_{j*6} + t_{1*6}
\end{aligned}
$$

It seems appropriate here to mention two of the limitations of this algorithm as presented.

1) The algorithm does not include a systematic clean-up of the code. This subject will be discussed in a later newsletter on variable subsumption and test replacement. [illegible] – Linear Function Test Replacement
2) The algorithm does not recognize the fact that all generated temporaries are themselves induction variables. Thus, in the above example, x and y might become induction variables after reduction in strength and a later instance of x+c might be reducible. The generalizations to handle this case will also be the subject of a later newsletter.

We have presented a very simple reduction in strength algorithm based on the principle of hashed temporaries in hope that it will be a first step toward more general algorithms.