SETL Newsletter No. 105 April 23, 1973

SETL Programs for a Basic Block Optimizer and an Extended Basic Block Optimizer

S. Marateck, J. Schwartz

In the following, we give SETL code for the basic block algorithm described in Cocke & Schwartz, pp. 328-332. In a subsequent section, this is generalized to an extended basic block optimizer algorithm. The data structures assumed, and the principal data structures used by the basic block optimizer, are as follows:

a. Code is represented by a tuple *code*, each of whose entries is a vector, whose components, in sequence, are *result value-number, operation-code, first input, second input, do-flag.* For indexed stores there is a third operation argument and a final component *third input* is present in this tuple. All other operations are assumed to be binary. The vector components just listed are given specific names in the following positional macro:

```
macro valno, op, in 1, in 2, do, in 3 endm;
```

b. Some entries in the code vector represent variables or constants. Such entries are assumed to have the following components: *value number, operation-code* (designating 'variable' or 'constant'), *hash table pointer.* If the entry represents a constant, then a fourth component, giving the *constant value*, is present. These components are named in the following positional macro:

```
macro valno, op, htptr, constval endm;
```

c. Four auxiliary maps are built up by the basic block optimizer as it acts:

i. *constfrmval* maps the value numbers of constants into their actual values;

ii. *valfrmconst* is inverse to *constfrmval;*

iii. *availcomps* is a map with three parameters, respectively designating an operation code and a first and second argument value number. If an operation with this description has been

performed before, *availcomps* gives the value number of the result.

iv. *opfrmval* maps a value number into the index of the first code item whose result has this value number.

v. *kind* maps each operation code into an integer in the range 1 to 5 designating one of the five main kinds of entries in the *code* vector: computation, simple assignment, indexed assignment, indexed load, and variable.

Several subroutines are used in the code which follows.

A. *valnum(i)* has as input the index of a code item representing a variable, constant, or operation. It returns the value number associated with this item, i.e. either the result value number or the value number assigned to a variable or constant. When a variable or constant is encountered for the first time within a block, *valnum* assigns it a value number; in the case of constants, appropriate entries are also made in *constfrmval* and *valfrmconst*.

B. *newvalno* issues unique value numbers, using an auxiliary counter *valnoctr* to do so.

C. *calculate* takes two constants and an operation code as input, and combines the constant using the operation to produce a compile-time result. Code for this subroutine, whose structure is obvious, will not be given.

Here follows the SETL code.

```
define  bblockopt(start,end);
/* SETL algorithm for basic block optimizer */
/* code, kind, constfrmval, opfromval,        and valnoctr are
   assumed to be global */
/* start and end delimit the basic block to be optimized */
constfrmval = nl; /*maps value numbers of constants into constants*/
valfrmconst = nl; /*maps constants into their value numbers*/
availcomps = nl;  /*gives result value number from operation
        and input value numbers */
opfromval = nl; /* gives operation producing value having given number*/
```

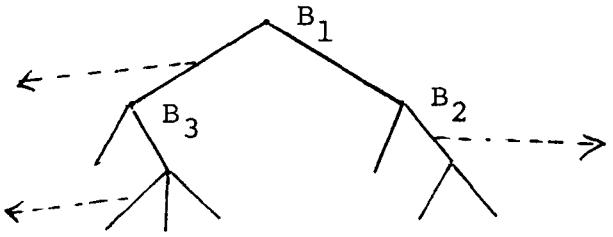
```
valnoctr = 0; /* auxiliary counter for issuance of unique
        value numbers */
(start ≤ ∀itmno ≤ end)
   item = code(itmno); /* get next operation item and
        its operation code */
   opn = op(item);
   go to <computation, simpassign, indexassign, indxload,
        variab>(kind(opr));
computation:
   il = in1(item); i2=in2(item); /* two inputs of binary operation */
   vl = valnum(il); v2 = valnum(i2);
        /* if necessary, force both inputs to have value numbers */
   if(constfrmval(vl) is cl) ne Ω
        and (constfrmval(v2) is c2) ne Ω
   then go to fold;;
procop:
   /* else/ if (availcomps(op,vl,v2) is oldval) ne Ω then
        do (code(itmno)) = f;
        valno (code(itmno)) = oldval;
   else
        availcomps(opn, vl, v2) =(newvalno( ) is opval);
        opfromval(opval) = itmno;
        do(code(itmno)) = t;
        valno(code(itmno)) = opval;
   end if;
   continue;
fold:
   c = calculate(opn,cl,c2); /* perform compile-time calculation*/
   if valfrmconst(c)  is cvalno ne Ω then
        valno(code(now)) = cvalno;
   else
        valno(code(now)) = (newvalno( ) is cvalno);
        constfrmval(cvalno) = c;
        valfrmconst(c) = cvalno;
   end if;
```

```
   do(code(itmno)) = f;
   continue;
simpassign:
   variable = in1(item); quantity = in2(item);
   valno(code(variable)) = valnum(quantity);
   continue;
indxassign:
   variable = in1(item); index = in2(item); quantity = in3(item);
   v2=valnum(index); v3=valnum(quantity);
   valno(code(variable)) = (newvalno( ) is newvarvalno);
   availcomps(indxftch,newvarvalno,v2) = v3;
   continue;
indxload:
   i1 = in1(item); i2=in2(item);
   v1= valnum(i1); v2=valnum(i2);
   go to procop;
variab:
   continue;
   end ∀ itmno;
   return;
   end;

definef valnum(itmno)
/* code is assumed to be global */
if valno(code(itmno)) is itemvalno ne Ω then
   return itemvalno;
else if op(code(itmno)) ne constop then /* case of variable */
   valno(code(itmno)) =(newvalno( ) is newval);
   opfromval(newval)   = itmno;
   return newval;
else /* treat the case of a constant */
   constvalue = constval(code(itmno));
   if valfrmconst(constvalue) is cvalno eq Ω then
        cvalno = newvalno( );
        valfrmconst(constvalue) = cvalno;
        constfrmval(cvalno) = constvalue;
```

```
   end if;
   valno(code(itmno)) = cvalno;
   return cvalno;
end if;
end valnum;

definef newvalno;
/* valnoctr is assumed to be global */
valnoctr = valnoctr + 1;
return valnoctr;
end newvalno;
```

2. Optimization of extended basic blocks.

We now describe an extended basic block optimizer. This algorithm processes sections of code ('extended basic blocks') in which each code item has a unique predecessor. The logical structure of such a section of code is as shown in the following figure.



Each solid line in this figure represents a basic block belonging to the extended basic block. The dotted lines represent transfers to sections of code not belonging to the extended basic block.

In optimizing extended basic blocks we shall use an optimizer algorithm much like that given in the preceding section; each time an *internal* branch (such as B_1,B_2,B_3 in the figure) is encountered, we (in effect) save the state of the optimizer's *availcomps* data set, optimize down one path from the branch,

and then return to the immediate pre-branch situation and optimize down the other branch. Note that to restore the pre-branch situation, we must not only drop from *availcomps* all computations performed after the branch but must also restore the pre-branch value numbers of all variables.

These two acts of logical restoration may be accomplished simply and efficiently as follows. On encountering a branch, we stack up each of its internal descendant blocks, at the same time associating with each descendant the immediate pre-branch value *vnpre* of the value number counter. Then later, when we return to process a basic block B whose optimization has been postponed, the value number counter will have reached a value which we may call *vnpost*. In processing B, we treat as irredundant any calculation whose associated value number lies between *vnpre* and *vnpost* , since all such calculations will have been performed in sections of code that do not precede B (logically) in the extended basic block.

Restoration of variable values is handled as follows. We associate a list of value numbers, rather than a single value number, with variables. The latest value number on the list is always used, unless this value number lies in the 'forbidden' range between *vnpre* and *vnpost*, in which case entries are dropped in order from this list with a value not exceeding *vnpre* is encountered (if the list becomes empty, a new value number is assigned).

In the algorithm which follows, we use the data structures assumed by the basic block optimizer described in the preceding section, plus a few more. The *code* array is allowed to contain items of type *branch*; the first input (i.e., component *in1*) of a branch item is a vector, giving all the possible targets of the branch (*in2* defines the branch condition, or, in the case of multi-way, indexed branches, the branch index value). A mapping *blockend(label)*, assumed to be available when extended basic block optimization is initiated, gives the final code item

of the basic block whose initial code item is *label*. If *blockend(label)* is undefined, the block starting at *label* is external to the extended basic block being processed.

Here is the SETL code.

```
define  ebbopt(start,blockend);
/* SETL algorithm for extended basic block optimizer */
/* code, kind, blockstack, constfrmval, valfrmconst,
   availcomps,opfromval and valnoctr are assumed to be global*/
/* the meaning of the parameter blockend is explained in the
   preceding text */
constfrmval = nl;  /* maps value numbers of constants into constants*/
valfrmconst = nl;  /* maps constants into their value numbers */
availcomps = nl;   /* gives result value number from operation
        and input value numbers */
opfromval = nl;    /* gives operation producing value having
        given number */
valnoctr = 0;      /* auxiliary counter for issuance of unique
        value numbers */
blockstack = <<start, blockend(start), valnoctr>>;
(while blockstart ne nult);
   <start, end, vnpre> = blockstack(#blockstack);
   blockstack(#blockstack) = Ω;
   newbbopt(start, end, vnpre, valnoctr+1);
end while;
return;
end ebbopt;

define newbbopt(start, end, vnpre, vnpost);
/* basic block subroutine for use with extended basic block
   optimizer */
include ebbopt(blockend);  /* the blockend map is required
   in this routine */
/* code, kind, constfrmval, opfromval, blockend, blockstack
   are valnoctr are assumed to be global */
```

```
/* start and end  delimit the basic block to be optimized */
(start ≤ ∀itmno ≤ end)
   item = code(itmno); /* get next operation item and
        its operation code */
   opn = op(item);
   go to <computation, simpassign,  indxassign, indxload,
                              branch, variab>(kind(opr));
computation:
   il = in1(item); i2=in2(item); /* two inputs of binary operation */
   vl = valnum(il); v2 = valnum(i2);
        /* if necessary, force both inputs to have value numbers*/
   if(constfrmval(vl) is cl) ne Ω
        and (constfrmval(v2) is c2) ne  Ω
   then go to fold;;
procop:
   /* else */ if (availcomps(op,vl,v2) is oldval) ne Ω andd
        oldval le vnpre or oldval ge vnpost then
        do(code(itmno)) = f;
        valno(code(itmno)) = oldval;
   else
        availcomps(opn,vl,v2) = (newvalno( ) is opval);
        opfromval(opval) = itmno;
        do(code(itmno)) = t;
        valno(code(itmno)) = opval;
   end if;
   continue;
fold:
   c = calculate(opn,cl,c2); /* perform compile-time calculation*/
   if valfrmconst(c) is cvalno ne Ω then
        valno(code(now))  = cvalno;
   else
        valno(code(now))  = (newvalno( ) is cvalno);
        constfrmval(cvalno) = c;
        valfrmconst(c) = cvalno;
   end if;
   do(code(itmno)) = f;
   continue;
```

```
simpassign:
   variable  = in1(item); quantity = in2(item);
   putvalno(variable, valnum(quantity));
   /* the subroutine putvalno(i,v), for which code is given
       below, updates the list of value numbers associated
       with the variable code(i), putting the value v at the
       head of this list */
   continue;
indxassign:
   variable = in1(item); index = in2(item); quantity = in3(item);
   v2 = valnum(index); v3 = valnum(quantity);
   putvalno(variable, newvalue( ) is newvarvalno);
   availcomps(indxftch, newvarvalno,v2) = v3;
   continue;
indxload:
   i1 = in1(item); i2=in2(item);
   v1 = valnum(i1); v2=valnum(i2);
   go to procop;
branch:
   targlist = in1(item); /* get list of target addresses */
   (∀st(j) ∈ targlist | blockend(st) is newend ne Ω)
        /* make entry on blockstack for later processing */
        blockstack(#blockstack+1) = <st,newend,valnoctr+1>;
   end ∀st;
   return; /* since current block is at an end */
variab:
   continue;

end ∀itmno;
return;
end;
```

```
define  putvalno(variable,newvalno);
include newbbopt(vnpre,vnpost);
/* value numbers not in the range vnpre to vnpost will be
   deleted */
if valno(code(variable)) is itemval ne Ω
   /* case of an old variable */ then
   (while itemval ne nult)
        if hd itemval is itemvalno gt vnpre or itemvalno lt vnpost then
           itemval = tℓ itemval;
        else
           valno(code(variable))=(if itemvalno le vnpre then<newvalno>
              else nult) + itemval;
           return;
        end if;
   end while;
   /* treat fallthru  case just as case of new variable */
end if;
valno(code(variable)) = <newvalno>;
return;
end putvalno;

definef valnum(itmno);
include newbbopt(vnpre,vnpost);
/* only value numbers not in the range vnpre to vnpost are
   acceptable */
if valno(code(itmno)) is itemval ne Ω then
   if op(code(itmno)) ne variable then return itemval;;
   /* in the case of a variable, itemval will be a list.
      we examine its leading elements, rejecting those
      which lie in the forbidden range from vnpre to vnpost */
   (while itemval ne nult)
      if hd itemval is itemvalno gt vnpre or itemvalno lt vnpost then
         itemval = tℓ itemval;
      else /* replace value list with edited version */
         valno(code(itmno)) = itemval;
         return itemvalno;
      end if;
```

```
   end while;
   /* on fallthru,  a new value number must be assigned */
   valno(code(itmno)) = <newvalno( ) is newval>;
   opfromval(newval) = itmno;
   return newval;
else /* case of a constant or a hitherto unencountered variable */
   if op(code(itmno)) ne constop then /* case of variable */
   valno(code(itmno)) = <newvalno( ) is newval>;
   opfromval(newval) = itmno;
   return newval;
else /* treat the case of a constant */
   constvalue = constval(code(itmno));
   if valfrmconst(constvalue) is cvalno eq Ω then
      cvalno = newvalno( );
      valfrmconst(constvalue) = cvalno;
      constfrmval(cvalno) = constvalue;
   end if;
   valno(code(itmno)) = cvalno;
   return cvalno;
end if;
end valnum;
```

The routine *newvalno* is unchanged from the preceding section.