SETL Newsletter # 170

Robert Dewar
Art Grand
Ed Schonberg
Len Vanek
April 22, 1976

Provisional Plan for the SETL Optimizer Interface

Section I Introduction

This newsletter specifies the input/output format of the SETL optimizer. Its purpose is to isolate two processes: the abstract specification of the optimizer and the concrete coding of the remaining portions of the SETL system. We will specify a number of data structures in this newsletter. It is important to realize that these in no way effect the design of the optimizer; they are merely an input/output medium. The optimizer should view them as a list of information received and information to be returned, with no relation to the optimizer's internal data structures. The details presented here are necessary if we are to proceed with the remainder of the SETL system before the optimizer design is finished. We begin by discussing the general organization of the SETL compiler.

1. The parser translates source programs to trees.
2. The semantic pass generates more detailed trees and determines the runtime representations of those variables which have been declared by the user. It creates 4 tables: a list of quadruples called *code*; a map from variables to their representations called *Reptab*; a map of constants to their values, called *Val*; and *stackmap*; a boolean map indicating which variables are stacked.
3. The optimizer adds information to the tables produced by the semantic pass.

   The changes it makes to the program fall into 5 categories:

a. Adding and deleting quadruples.

b. Setting flags in the quadruples to indicate destructive use conditions, etc.

c. Filling in *Reptab* for undeclared variables. This is somewhat complicated since declared variables have a single type throughout the program while undeclared variables may have several types.

d. Preparing a new table *Equivtab*; which indicates variables which may optionally use the same storage locations.

e. Modifying *stackmap*.

4. The code generator forms a new sets of quadruples which is suitable either for interpretation or machine code generation.

Section II. Definitions

In this section we present formal definitions for the various tables, etc.

Definition 1: a *program variable* is a pair <name, scope>. In the LITTLE implementation these pairs are represented by pointers into a separate table.

Definition 2: a *scope* is a pair <scope name, extflag> where *exflag* indicates whether a variable is external.

Definition 3: During execution of a SETL program a variable may receive a value, become dead and receive a value of a new type. We call this new value a *reincarnation* and say that such a variable has several *incarnations*. Each *incarnation* can be thought of as a separate variable with a static type. This allows for simple description of type information.

Definition 4: An *unoptimized program* consists of

1. *Code*, a vector of quadruples.
2. *Reptab*, a map from *program variables* to *representations*.
3. *Val*, a map from *program variables* to *values*, defined only on program variables which are constants.
4. *Stackmap*, a boolean map on program variables.

Definition 5: a *"quadruple"* is a 13 tuple with the following fields:

OPCODE	an integer demoting an operation
ARG1	a program variable which is the output of the operation
ARG2, ARG3	program variables used as inputs.
LIVE1, LIVE2, LIVE3	these fields indicate whether their corresponding arguments are definitely live, definitely dead, or undetermined.
DUSE	this indicates whether ARG2 can be destructively used. It has three values: yes, no, and must be checked at runtime.
SETSHARE	indicates whether the output's share bit must be set at runtime.
CHECKR	invokes type checking on the result.
STMTNO	statement number inherited from the source program.
NLEV	indicates the number of loops surrounding the operation. This is used to determine whether in-line code or a library call is appropriate.
NEXT	a pointer to the next quadruple.

In the optimizer algorithms we represent a quadruple as <opcode, output, input1...inputn> with any number of inputs.

Definition 6: a *representation* indicates how an object is stored. It indicates both type and basing information.

A representation is defined to be one of the following:

1. a primitive type or the union of several primitive types. The primitive types are

   a. bits.
   b. blank atoms.
   c. characters.
   d. integers.
   e. labels.
   f. procedures whose number of arguments and returned value type is unknown.
   g. reals.
   h. tuples whose element types are unknown.
   i. sets whose element types are unknown.

2. an element of a program variable *PV*
3. a set whose elements have type R_1, and whose average size is n or unknown.
4. a function with argument types R_1 thru R_n returning R.
5. a subroutine with arguments R_1 thru R_n.
6. a map from R_1 to R_2.
7. an smap from R_1 to R_2.
8. an amap of R_1's.
9. a known or unknown length tuple whose members are R_1's. In the case of a known length tuple R_1 may be a tuple of representations.
10. an aset of R_1's.

Representations are stored in the map *Reptab* which has the following fields:

RKIND	an integer from 1 to 10 indicating one of the seven rules above.
RMEMB1	the member representation. This corresponds to R_1 above.
RMEMB2	corresponds to R_2 above.
RBASE	the program variable on which something is based. corresponds to *PV* above.

RPRIM a 9 bit string corresponding to the primitives a through g.

RSAFE indicates that the representation is known to be correct and need not be checked at runtime.

RNO1 size of a set, tuple, or string, number of arguments or lowest value of an integer.

RNO2 maximum value of an integer.

Definition 7: *Val* is a map from program variables to their values. It is defined only on constants. In the implementation *Val* will be restricted to constants whose values are integers, bits, characters, labels and procedures. The value of labels, functions and subroutines is a code index.

Definition 8: Equivtab is a set of sets of program variables which may optionally share storage.

Definition 9: *Stackmap* is a boolean map on program variables indicating which variables are stacked on entry to a procedure.

Section III. The Quadruples

In this section we give a list of the quadruple opcodes plus descriptions of a few complex code sequences. The quadruple operations fall into two categories:

1. Quadruples which correspond to executable code. These quadruples have opcodes with the prefix 'op'.
2. Dummy operations inserted into *code* to simplify valueflow analysis. These opcodes begin with 'aux'.

Various operations use more than two inputs. For these operations, ARG1 contains the result, ARG2 the first input, and ARG3 the number of inputs minus 1. The remaining inputs appear in 'OP-PUSH' quadruples just prior to the operation. These correspond to pushes onto the runtime stack.

Code Sequences:

We present detailed code sequences for the more complex operations. Quadruples are shown as

<opcode, output, input1, input2...input n>

Each procedure begins with an entry block containing dummy assignments to its parameters. Each procedure has a temporary *rtemp* which is used for the returned value. It has an exit block which begins with a generated label *exitlab* and contains a dummy assignment of *rtemp* to itself.

The statement

    return x;

is translated as

    <OP-RET, *rtemp*, *exitlab*, x >

the optimizer treats this as

    rtemp = x;
    go to exitlab;

Note that the statement

    return;

is a macro for

    return om.;

y = f(x_1,...,x_n) is treated as

<auxarb, t_1, f >
<auxtl, t_1 >
<auxtl, t, >
.
.   } n aux-tail instructions
.
<auxtl, t_1 >
<opof, y, f, x_1,...,x_n >
<aux-oralt, x, y, t, >

f(x_1,...,x_n) = y becomes

(1) <aux-tup, t_1, x_n, y>
<aux-tup, t_1, x_{n-1}, t_1>
.
.
.
<aux-tup, t_1, x_1, t_1>
<aux-with, t_1, f, t_1>
<op-sof, f, x_1,...,x_n, y>
<aux-oralt, f, f, t_1>

f {x_1,...,x_n} = y has a similar treatment with
(1) replaced by

<aux-arb, t_2, x>
<aux-tup, t_1, x_n, t_2>

f [x_1,...,x_n] = y is translated as

<aux-arb, t_1, x_1>
<aux-arb, t_2, x_2>
.
.
.
<aux-arb, t_n, x_n >
<aux-tup, temp, t_n, y>
<aux- tup, temp, t_{n-1}, temp>
.
.
.
<aux-tup, temp, t_1, temp>
<aux-with, temp, f, temp>
<op-sofb, f, x_1,...,x_n, y>
aux-oralt, f, f, temp>

GROUP 1: EXECUTABLE INSTRUCTIONS.

BINARY OPERATORS

OP¬ADD	+
OP¬AND	AND.
OP¬CC	CONCATENATION
OP¬DIV	/
OP¬EXP	**
OP¬EQ	EQ.
OP¬GE	GE.
OP¬GT	GT.
OP¬IMP	IMP.
OP¬IN	IN.
OP¬INC	INCS.
OP¬LE	LE.
OP¬LESS	LESS.
OP¬LESSF	LESSF.
OP¬LT	LT.
OP¬MAX	MAX.
OP¬MIN	MIN.
OP¬MOD	//
OP¬MULT	*
OP¬NPOW	NPOW
OP¬OR	OR.
OP¬SUB	-
OP¬XOR	EXOR.
OP¬WITH	WITH.

UNARY OPERATORS

OP¬ABS	ABS.
OP¬ATOM	ATOM.
OP¬ARB	ARB.
OP¬BITR	BITR.
OP¬BOT	BOT.
OP¬DEC	DEC.
OP¬FIX	FIX.
OP¬FLOAT	FLOAT.
OP¬NELT	NELT.
OP¬NOT	NOT.
OP¬OCT	OCT.

```
OP¬POW        POW.
OP¬RAND       RANDOM
OP¬TOP        TOP.
OP¬TYPE       TYPE.
OP¬UMIN       UNARY -

MISCELLANEOUS

OP¬END        S(I:)
OP¬NEW        NEWAT.
OP¬READ       READ
OP¬SET        SET
OP¬STOP       STOP
OP¬SUBST      S(I:J)
OP¬TUP        TUPLE
OP¬WRITE      WRITE

MAPPINGS
OP¬OF         F(X)
OP¬OFA        F≤X≥
OP¬OFAN       F≤X1,...XN≥
OP¬OFB        F[X]
OP¬OFBN       F[X1,...,XN]
OP¬OFN        F(X1,...,XN)

ASSIGNMENTS
OP¬ARGIN      ARGUMENT IN
OP¬ASN        A=B
OP¬SOF        F(X)=Y
OP¬SOFA       F≤X≥=Y
OP¬SOFAN      F≤X1,...,XN≥ =Y
OP¬SOFB       F[X]=Y
OP¬SOFBN      F[X1,...XN]=Y
OP¬SOFN,       F(X1,...XN)=Y
OP¬SSUBST     S(I:J)=Y

CONTROL STATEMENTS
OP¬CALL       SUBR CALL
OP¬FCALL      FNCT CALL
OP¬GO         GOTO ARG2
OP¬IF         IF ARG2 GOTO ARG3
OP¬IFINIT     IF INITFLAG GOTO ARG2
OP¬IFNOT      IF NOT ARG2 GOTO ARG3
OP¬NEXT       ∨ ARG2 → ARG3
```

```
OP¬NEXTD     NEXT ELEMENT OF DOMAIN
OP¬RETASN    RETURN ASSIGNMENT
OP¬RET       RETURN


GROUP 2. AUXILIARY OPERATIONS
AUX¬ARB      DUMMY ARB. OPERATION
AUX¬ASN      DUMMY ASSIGNMENT
AUX¬SET      DUMMY SETFORMER
AUX¬TL       DUMMY X(2) EXTRACTION
AUX¬TUP      DUMMY TUPLE FORMER
AUX¬WITH     DUMMY WITH.
```