SETL Newsletter Number 45 July 19, 1971

Semi-Local SETL Optimization: Dave Shields

Introduction:

In this newsletter we briefly describe optimization of SETL, emphasizing the semi-local optimizations which should be realizable in the near future and optimizations related to the SETL implementation. We include examples, and suggest a schedule for optimization efforts for the next few months.

The central issue in SETL optimization is, of course, the optimization of the set-theoretic operations - set membership, functional application, etc. In addition, a lower level of optimization may be distinguished - this includes proper choice of basic support operations, design of a "SETL machine", etc. Any particular optimization may be deferred as long as a level of performance capable of supporting ongoing development work is maintained.

The types of lower-level optimization that may be necessary, for example, minimization of dynamic checking of types, and proper dynamic use of storage, are not peculiar to SETL, but are also applicable to languages such as PL/I, ALGOL 68, and BALM.

The second implementation of the SETL primitive operations in BALM is almost complete. The first implementation used BALM 3, represented sets and lists, and was completely interpretive. The second implementation represents sets as vectors of lists using hashing to index the vectors. The execution is also interpretive. The next implementation to follow will be compiler based, and some optimization will be done to eliminate needless tests for type, and to eliminate needless formation of sets. These optimizations are non-recursive in nature, and amount essentially to improved code generation.

The fourth implementation will make use of some global flow analysis and more extensive global optimizations.

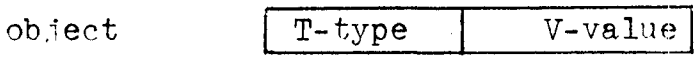
The tentative schedule for these efforts is as follows:

(1) September - Second implementation available, first efforts at code generation.
(2) January - SETL compiled, not interpreted; local optimization of set-former and short loops.
(3) Sometime later - fourth implementation, global program analysis to extend scope of available optimizations, multiple representations for sets.

I. Suppression of type-testing for constants and more general optimization of type-testing.

A survey of some optimizations:

SETL, BALM, and ALGOL 68 are examples of "type and value" languages. By this we mean that, at the implementation level, the fundamental data object is of the form



The type-field T is typically 3-6 bits in length, and the value field is 18-24 bits long. We consider the type field as either a small integer or a bit-string of implementation-defined flags. The value-field is usually a pointer to the implementation representation of the value of the object; for example, if the type-field is "set", then the value-field might address the top node of a tree containing the set members. However, certain objects, such as booleans or integers of fixed precision, may require no more bits to represent their value than are required for an address, and so their value may be stored directly in the value-field.

The type-field is typically used in two ways:

(a) to indicate how to interpret the value field, i.e., to determine if V is an address or the value itself; or

(b) to determine if the object is a "legitimate" operand, and if so, which form of operation to apply. For example, if "plus" is defined as addition for integers and catenation for strings, the type-field must be checked to determine which form of "plus" to use.

Usages of the form (a) are not common; the necessary interpretation is usually contained in the code implementing the primitive operations on the objects; for example, the code for integer addition might assume that the value-field contains the value of the integer. If such usages are buried too deeply, or obscurely, within the implementation, then it may be difficult to extend the language, for example, to allow integers of varying precision.

We now discuss the aspects of type-field particularly relevant when we are "compiling" SETL. This issue includes the isolation of tests on type so that redundant tests can be eliminated, the run-time scheduling of type tests, and the implementation of booleans.

Consider the algorithm for "plus" mentioned above, defined as addition for integers, catenation for strings, and Λ otherwise:

```
definef  plus(a,b); return ifx
       isinteger(a)?
  isinteger(b)?     ischar(a)?
=intadd(a,b),=undef,ischar(b)?          isbit(a)?
                =charadd(a,b),=undef,  isbit(b)?=undef
                                     =bitadd(b),=undef;
intadd:...code for add...;
bitadd;
...
end ifx;; end plus;
```

Observe that the calculation of plus(a,1) involves the needless test that "1" is an integer; "1b+4b" involves the redundant tests that the arguments are neither integers nor bit strings; and "1b+2" involves the needless tests that "1b" is neither an integer nor a character string (in fact, we know the result is Λ since "plus" is defined only for arguments of the same type). If the code for "plus" were written using "if...then..." statements instead of the ifx form, and if "plus" contained the code for the subroutines, intadd, bitadd, charadd, instead of the procedural form shown, it might be even more difficult to note the test redundancies mentioned above.

The examples show that the type-checking contained in the implementation routines should be "exposed" to the compiler; this allows the compiler to eliminate needless tests if operand types are known at compile time. A standardized implementation of type-tests of this sort also makes it easier to add new-object types to the language.

A suggestion concerning optimization of type-checking:

We can express the type-checking of operands in a tabular manner as follows:

⟨name, numargs, sametypes, typelist, errcase⟩,

in which name is the function name, numargs is the number of arguments, sametypes is true if all arguments must be of the same type and false otherwise; and typelist is a set of tuples of the form

⟨type, action⟩,

when action is either the name of the procedure to use if the

operands have type typ or a code skeleton. The last entry errcase is of the same form as action, and indicates what to do if no tuple in the typelist begins with the type of the operands. For example, for "plus" we obtain:

⟨plus, 2, true, {⟨integer, integeradd⟩, ⟨bit, bitadd⟩, ⟨character, charadd⟩}, undef⟩.

Algorithms for code-generation using such type information will follow in a later SETL newsletter.

A suggestion for the dynamic scheduling of type tests:

The methods discussed in the preceding paragraphs deal only with the static analysis of the program; we assumed no knowledge of the relative frequencies with which various parts of the program are executed, or of the structure of the data input to the program. However, in most SETL programs, the type of a given variable will not change during program execution. For example, in:

if a+b gt c then...

it is unlikely that a,b,c will have both integers and strings as their value during execution. Thus, if a,b,c are all bit strings, then every calculation of "a+b", involves the essentially needless tests that a and b are neither integers nor character strings.

Since the compiled code must contain tests for all the admissible types, the best we can hope for is to arrange the tests so that the correct test is made first. This can be accomplished as follows.

(a) Compile a special branch-on-type function, of variable length, which contains a list of valid types and addresses of corresponding code (note that the list entry has the standard form, i.e., we use type-field for valid types, and value-field for code address).

(b) Compile the calls to various subparts (e.g., intadd). The labels of these subparts are the addresses contained in the branch-instruction. Note that the instruction is compiled as an "initialized" branch-on-type instruction. The first time the instruction is performed, the tests are performed and the list in the opcode is altered so that the successful test is first, the initialization flag is turned off, so that subsequent executions of the instruction require just the linear search of the type-address list.

The "branch-on-type" instruction may be implemented on the BALM machine shortly.

II. Optimization of set-operations.

In this section we discuss the semi-local set optimizations to be included in the third SETL implementation.

The evaluation of certain expressions containing the SETL set-former may not require the formation of the set described. In particular, the optimizer should recognize the following forms, and compile code replacing the set-former by a loop:

(1a) $\exists\{e(x), x \in a \mid c(x)\}$ into

(1b) elm=Λ; ($\forall x \in a$), f c(x) then elm=x; quit;; end $\forall$x;

(2a) $y \in \{e(x), x \in a \mid c(x)\}$ into

(2b) mem=false; ($\forall x \in a \mid c(x)$) if y eq e(x) then mem=true;quit;; end $\forall$x;

Note that incorrect code may result if the evaluation of either c(x) or e(x) has "side-effects". For example, consider

(3) a=$\{1,2,3\}$;y=$\exists\{g(x), x \in a \mid x$ gt 0$\}$;...

```
definef g(x); external times; times=times+1; return 1/(x-2);
    end g;
```

Note that if we compile (3) as (1b) then times will be incremented once, and an otherwise fatal error that might result when x is 2 may be missed. Either we can change the definition of the set-former to require the programmer to expose such side-effects, or, more appropriately, we compile (1a) in the form (1b) only when no function calls are involved, or else compile tests to determine if functions involved are sets or programmer-defined functions, and branch to the respective short or long form accordingly.

One can also consider

$$(4a)\quad \#\{e(x), x \in a \mid c(x)\}$$

In many cases, we may as well form the set appearing on the right, since potential members must be tested for duplication. If, however, e(x) is known a priori to be one-one, so that no two potential candidates can be equal, then (4a) can be compiled as

```
(4)  num=0; (∀x∈ a | c(x)) num=num+1; end ∀x;
```

In addition, we can omit the test for element duplication in any instance of

$$\{e(x), x \in a \mid c(x)\}$$

in which e(x) is one to one.

Since the verification that an expression e(x) is one-one poses a difficult problem in theorem proving, for the present the compiler will only recognize this property if e(x) is of the form e(x)=x, or e(x)= tuple with x as one component.

III. Elementary constant-noticing.

Sets and tuples defined by explicit enumeration of their elements should be separated into their definitely-constant and possibly-varying parts. For example,

$$\{a, b+1, 10, 20\}$$

is to be considered as

$$\{a, b+1\} \text{ union } \{10, 20, 30\},$$

where the set on the right need only be formed once, at compile time. This optimization is particularly relevant for tables and computed go-tos, e.g.

goto $\{\langle 1, \text{lab1}\rangle, \langle 2, \text{lab2}\rangle, \langle 3, \text{lab}\rangle\}$ compindex;

IV. Miscellaneous.

Other special forms and tests should also be recognized by the compiler. These include.

hd[set],

if set is a function-set, i.e., the domain of the function set. This is only feasible if the implementation of sets "groups" tuples with identical first components so that an appropriate fast routine can be written.