SETL Newsletter # 148

Technical and Human Factor Improvements for the Fully Compiled SETL System.

Ed Schonberg, A. Stein
April 21, 1975

SETL1, the first fully compiled implementation of SETL, is nearing completion. This system uses a BALM-to-LITTLE translator, written in BALM, to produce LITTLE source code, which is then compiled as a LITTLE program and executed in the environment of the run-time library. The translator is itself interpreted by the BALM interpreter, writtin in LITTLE, which also utilizes the SRTL. The full system consists of the following modules:

a) SETLA: A lexical scanner and syntactic analyzer. SETLA is written in LITTLE. Its output consists of various token tables and a sequence of calls to generators.

b) TREEWLK: A series of treewalking routines, written in BALM, which build BALM parse-trees using the output from a).

c) TRANS: The translator proper, which replaces the code generators of the BALMSETL system, so that LITTLE code is emitted (instead of extended MBALM code).

d) BALMINT: A BALM interpreter, which uses a portion of SRTL, and executes b) and c).

e) The LITTLE compiler.

f) SRTL.

As it stands, the system is unwieldy both in terms of storage requirements and execution time. This is due to the structure of TRANS, and to some extent to the presence of LITTLE as the target language for TRANS. This later constraint is likely to be with us for some time. We will consider in what follows some ways of streamlining TRANS to remove some of its more glaring bottlenecks. In particular, the interface between TRANS and LITTLE will be modified and made more modular. This should simplify addition of further SETL features, and eventual modification of the target language.

The procedures in TRANS are for the most part in 1-1 correspondence with the code-generating procedures of the BALM compiler. For example to the procedure GWHILE, which emits code for

a while-loop, corresponds in TRANS the procedure LGWHILE, whose structure is virtually identical (in terms of generation of labels, tests and jumps). The code being generated is first assembled into a list (the global variable LIST2 in both cases). In the BALM compiler, procedure ASS. keeps track of the last element of LIST2 and appends to it successive MBALM opcodes. In TRANS this role is taken by procedure ENCODE. However, the arguments of ENCODE are not simple opcodes but may be constants, variable names, addresses (in the form of stack locations, global identifiers or compiler-generated temporaries) or code fragments in the form of lists of any of the above. In order to linearize this code stream, ENCODE must be recursive. When the code for a full SETL procedure has been generated, procedure FORMAT processes LIST2, replaces identifiers by their names, constants by their external representations, etc. and outputs the LITTLE source to an intermediate file, ready for input into the LITTLE compiler.

The above description should make the following inefficiencies apparent:

a) LIST2 must be kept in core until a full procedure is processed. The storage requirements for compilation of the simple SETL programs thus become comparable with those of executing very substantial programs on the current interpretive system (100K of dynamic storage or more).

b) ENCODE must process every single token of the LITTLE code being produced. It is called repeatedly with long lists of stereotyped arguments (parentheses, separators, keywords) with the corresponding expense in stack manipulation; because the LITTLE code is generated piecemeal through TRANS, ENCODE must be recursive to be able to process lists and trees.

c) Formatting is also done in dynamic storage, interpretively, and makes little use of the existing SETL I-O.

The following modifications suggest themselves:

1) ENCODE, which is the bottom-level procedure of TRANS, will be coded in LITTLE and made into a BALM primitive.

2) LIST2 will be formatted on the fly by ENCODE, and written directly to an intermediate file.

3) TRANS will not assemble any code fragments. All string manipulation will take place within ENCODE. The only objects manipulated by TRANS will be BALM identifiers, integers, and addresses. These will be created as objects of special types within SRTL, and recognized by ENCODE, which will emit the correct LITTLE form.

4) The FORMAT procedure will be eliminated altogether.

5) The code produced by ENCODE will be a series of macro invocations, that will be expanded by the LITTLE front-end in the usual fashion. The first argument in every call to ENCODE will be a macro name. The required macro definitions will be added to the MACRO deck of SRTL.

6) Variable addresses will be encoded as SRTL blank atoms. They are generated internally as indices, and expanded as offsets from global pointers. The value field of each blank atom will contain two subfields: one will hold the value of the index; the second (a 3-bit field) will specify the type of address being encoded. (Note that the remaining 30 bits should be sufficient to encode any conceivable compiler-generated offset.)

The types required correspond to:

a) global variables, i.e. symbol table entries
b) stack variables
c) stack temporaries
d) procedure arguments
e) generated labels.

7) Finally, one additional type must be added to SRTL to handle SETL integers. This is due to the fact that the output of TRANS contains both SETL integers (generated by constants in the SETL source program) and LITTLE integers, used in loops, address offsets, etc.; however, both types are represented internally (i.e. in the interpreter) as SETL integers. "True" SETL integers will therefore be flagged as a separate type. ENCODE will generate the necessary call to GENINT to rebuild these integers at execution time.

This underlines the very inefficient but currently unavoidable way in which constants are handled: any SETL atom is scanned and recognized by SETLA, transformed into its SETL internal form for TREEWLK, then converted back to its external representation, printed, rescanned and reconverted at execution time. A more rational scheme would involve some sharing of dynamic storage between TRANS and the execution time environment.

8) Some savings can easily be achieved for short objects. For these (integers and strings) ENCODE will produce directly the 60-bit constant representing the item. As set lists are never created during compilation, the link-field of such items will always be blank, and the use of the bit-representation is safe.

Incrementality

Experience to date with the SETLA interpretive system (and previous experience with SETLB) indicate that the incrementality provided by the SAVESETL procedure is extremely useful, and most users avail themselves of it as soon as their programs grow beyond a certain size (say 200 source lines). We propose to create a similar feature for the SETLA translator system. As described here, it will provide most of the services of the SAIM save-resume operation, without its full elegance and flexibility however.

In the interpretive system, execution of the SAVESETL procedure produces a SAVEFILE which holds the contents of heap, stack, and symbol table. As the heap contains all compiled code (including the treewalk routines and code generators) and the stack contains all pointers associated with linkages active at the time SAVESETL is executed, the SAVEFILE can support any subsequent compilation and execution (indeed, both of these are identical in SAIM).

In the translator system, compilation and run-time environment are distinct. During the latter, dynamic storage contains only the user's symbol table and associated values. Code (produced by the LITTLE compiler) is now loaded by the host system, and does not reside in dynamic storage. A machine-language patch must be used to link it to the stack (to implement parameter passing and

recursion--see LITTLE newsletter no. 31). The symbol table entries corresponding to user procedures now hold absolute machine addresses (entry points to executable code) instead of heap pointers to garbage-collectible blocks.

By contrast, the compile-time environment contains a standard BALM savefile, whose heap holds the packed MBALM code for compiler, treewalk routines and modified generators. Its symbol table links to all of these, and holds in addition the user's variables. It is only this latter portion of the symbol table which is shared by the compile-and execution time environments. Incrementality requires that this portion of the symbol table, and the compiled code, be transmitted across jobs. Incrementality of compilation can therefore be insured as follows:

a) By saving the binary output of the LITTLE compiler, in the form of a user library (in KRONOS, by DEFINE-ing the LGO file as a permanent file)

b) By saving in addition the user's symbol table, simply as a list of names.

When compiling a subsequent program, this symbol table will be read-in by the initialization procedure of the translator, so that the invocation of precompiled user procedures can be recognized, and name-resolution accomplished. (Note that this scheme will be compatible with the full SETL namescoping scheme, or the subset proposed in newsletter 128.)

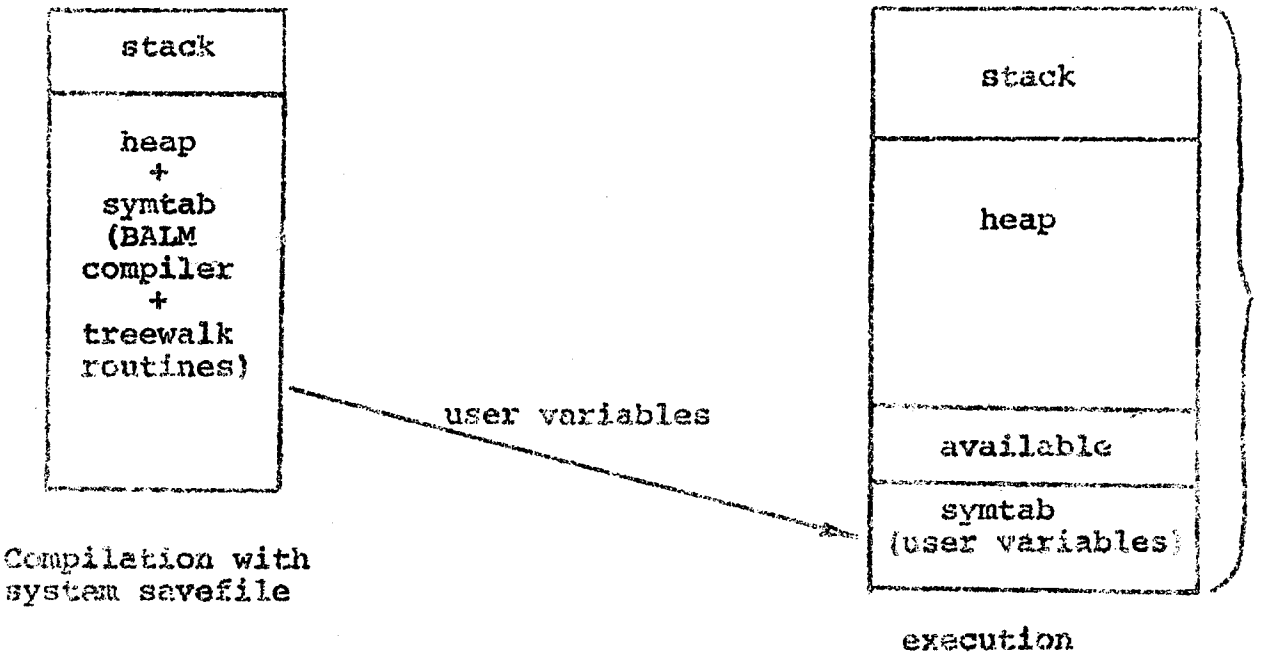
The translator will add to this symbol table the new user variables, and transmit this extended list to the execution-time environment. The user library will be loaded together with SRTL, and the initialization phase of SRTL will link the symbol table entries to the new user code, the user library and SRTL.

Incrementality of execution can be obtained by a simple modification of this scheme. Instead of saving the user's symbol table, we can save the full state of dynamic storage: heap, stack and symbol table. The compilation phase will collect the information it needs from the save-file, expand the symbol table, and rewrite into the savefile, which will then be copied into

dynamic storage at the beginning of execution. This scheme of course requires that the expansion of the symbol table should not alter the organization of dynamic storage. This means that the symbol table will now have a maximum fixed size, and a fixed location (at the base of the heap). (See Fig.)

An additional constraint (from the point of view of a user familiar with the current SAVESETL feature) is that a save will only be allowed from the main program, i.e. when no user procedures are active. This insures that no machine addresses need be saved, as the stack will contain only the base-environment blocks of each user procedure.

## NORMAL EXECUTION




## EXECUTION WITH USER SAVEFILE