SETL Newsletter # 154

Ed Schonberg
July 24, 1975

Timing considerations for the

SETL translator system.

I. Introduction.

We report here on a series of measurements which compare the SETL interpretive system (SETLA) with the translator system (TSETL). TSETL has successfully executed all the programs in the SETL test package (which constitutes an informal "acceptance test" for this new SETL system) and can be considered reasonably complete (i.e., as having reached the point where "debugging" becomes "maintenance", however without a sharp decrease yet in the frequency of bug-finding.)

Timing tests were performed to provide an estimate of the relative efficiencies of SETL and LITTLE. The ratio in execution speed between the two represents the maximal gain attainable by an optimizer, and these figures will therefore orient some of our work on the global optimizer currently planned for the next SETL system.

Five algorithms were used for these tests. Coded first in SETL, they were then rather carefully hand-transcribed into LITTLE. This transcription did not attempt to attain quite the sophistication of the SETL run-time library, and some obvious short-cuts were taken to simplify the coding (for example, no hashing was used, and sets were encoded as arrays, sorted or unsorted). This exercise in "two-stage programming" went remarkably smoothly (considering the complexity of some of the algorithms included), and gives confidence in the use of a high-level specification language to create a production-level program.

II. Algorithms used.

1) As a "worst case" comparison, matrix multiplication was chosen. In the SETL version of the algorithm, two alternate representations were used for matrices:

a) A set of ordered triples.

b) A tuple of tuples.

This is clearly an algorithm for which SETL is particularly ill-suited: in a) because of the way long tuples in sets are represented internally in SRTL (adding 2 levels of indirect addressing); in b) because of the number of useless copy operations triggered by retrieval of each row of the matrix. In contrast, the LITTLE version benefits from the excellent register allocator of the LITTLE compiler and produces high-quality code for tight loops.

2) Hoare's "heapsort" algorithm. This algorithm is totally static i.e., makes no space allocation and leads to no garbage collection; the codes in SETL and LITTLE are identical. The ratio SETLA/TSETL is a good measure of the "interpretive overhead" of SETLA. The ratio TSETL/LITTLE reflects the cost of indirect addressing and off-line SRTL invocation inherent in the LITTLE code produced by TSETL.

c) The Huffman encoding algorithm (See *On Programming* II, p. 148). The coded message was represented internally as a packed bit-string both in SETL and LITTLE. In the LITTLE version, the internal representation of individual characters was used to index directly into the map of codes, a machine level optimization which is probably beyond the reach of any automatic optimizer. A separate test of the tree-building procedure used by the Huffman algorithm affords a measure of the expense of SETL recursive linkages vs. LITTLE hand-coded stack manipulation.

d) An algorithm to transform an arbitrary BNF grammar to Chomsky normal form (see O.P. II, p. 176). In the LITTLE version, productions, terminal, and non-terminal symbols were kept as arrays, and membership tests were simply linear searches. A simple compaction procedure was used to remove erasable productions from the normalized grammar.

e) An algorithm for solving the maxflow problem (see O.P. II, p.120). The LITTLE version represented sets of edges as ordered arrays, on which membership tests were performed as binary searches.

Results:

Timings (in CPU. secs)

Algorithm	LITTLE	SETLA	TSETL
Matrix multiplication (15 ×15)	.08	a) 20 (10 secs. garbage collection)	6.7
		b) 34 (15 secs. garbage collection)	10.8
Heapsort (100 elements)	.022	5 (1.7 sec. garbage collection)	1.4
Huffman encoding (10 lines)	.21	14 (2 secs garbage collection)	4.5
building Huffman tree for 256 random frequencies	1.7	87	32
Chomsky normalizer (17 productions → 74)	.58	14	3.2
Maxflow (100 nodes, 132 edges)	1.04	22.4	7.3

Discussion

The following order of magnitude figures emerge:

a) TSETL is 2 to 5 times faster than SETLA. This factor corresponds to the interpretive overhead of SETLA. It is no larger than 5 (as might be expected at first) because for all but the simplest programs, both systems spend most of their time executing SRTL procedures. This is clearly the case in programs involving large amounts of set manipulation.

b) SETLA is 20-200 times slower than LITTLE. For TSETL the figure is 7-50. The larger figure coincides with earlier estimates of system efficiency. The lower one is surprisingly favorable to the SETL system, and obtains in cases where set manipulations (membership tests, inclusion, etc.) are used heavily (the last three algorithms tested). This reflects the very careful coding of SRTL, and indicates that, for "one-shot" programs of sufficient complexity, it is already reasonable to use SETL and avoid the tedious debugging that the equivalent LITTLE program would entail.

c) The range of improvement which can be expected from a global optimizer is therefore surprisingly narrow. In many cases, determining that a set can be treated as a static array (i.e., that it need not be accessible to the garbage collector, or that it can be allocated on the stack) will provide the greatest pay-off. (This requires of course full type-determination, to ascertain that all members of the set are "short" objects containing no heap pointers). The elimination of redundant copy operations (which requires full value flow analysis) will provide the other important improvement. Better peephole optimization, global optimization of SRTL, use of membership and inclusion relations among program variables to produce optimal data-structures, and other such sophisticated devices may have little effect on system performance, compared with that of the first two mentioned.

Overall TSETL performance.

The figures given above correspond to execution time. The compilation time overhead of the current TSETL system makes it actually less interesting to the SETL user who is not running "production" programs in SETL, ie. program with execution times >300 cpu secs. The reason for the compilation overhead are twofold:

a) The "compiler" itself, i.e., the code generators which emit LITTLE source code, are written in BALM and are themselves executed interpretively (in fact by the same interpreter which runs SETLA). Because of the high semantic level of the MBALM primitives vis-a-vis of LITTLE, several lines of LITTLE are produced in TSETL for each equivalent MBALM opcode. The formatting of these lines and the need to refer explicitly to stack locations (while MBALM opcodes work implicitly on the stack) makes the code-generation phase 2-3 times slower in TSETL than in SETLA.

b) The LITTLE code produced must be compiled with the MACRO and START decks of SRTL, adding 1500 card images to every compilation. In addition, code expansion is of the order of 50 lines of LITTLE/line of SETL source, so that typically thousands of lines of LITTLE must be compiled. The code expansion factor mentioned above is to be expected, given the difference in semantic levels between the two languages; in fact the LITTLE code currently produced by TSETL is of reasonable quality, i.e., has no obvious redundancies or wasteful register manipulations. The simple fact is that LITTLE is in a sense too high-level a language to be the target of the present code generator (which uses a small subset of the LITTLE language features). However, it would be a major task to redesign the interface between TSETL and LITTLE.

In the meantime, it will be more practical to use SETLA for debugging runs, small experiments, and language learning, and to reserve TSETL for sizable "production" runs.

Typical compilation rates. (cpu secs)

	SETLA	TSETL
matrix multiplication	11	LITTLE generation: 23 LITTLE compilation: 22
Chomsky normalizer	50	L. generation: 127 L. compilation: 62