SETL Newsletter Number 156 August 1, 1975

The Next Phase of Our Work J. Schwartz

The fully compiled version of the SETL system (SETLC) is now functioning adequately (though problems still exist; see below). Our main effort from now on will be directed at the global optimization of SETL. Toward this end the following has already been done:

(a) A large variety of peephole optimizations have been installed. Our experience with these is disappointing; speed gain is small. This makes it clear that purely local optimization of SETL cannot improve its speed much.

(b) Extensive, but still quite unpolished, SETL code for the various forms of global analysis described in 'Optimization of Very High Level Languages' (Jour. Prog. Langs., v. 1, # 2,3) has been developed.

(c) Dave Shields' work on global optimization of LITTLE is going forward. When complete, this should improve the speed of our whole system by about 20 %.

(d) A semi-automatic data structure choice scheme has been outlined in NL 151.

Two related but distinct possibilities now lie open to us:

i. To implement the various 'easy' fully automatic optimizations which can be based on the analysis (b). The main opportunities here are: test elision based on typefinding, elimination of unnecessary copying, a better treatment of integers based on 'short integer' detection, and various subroutine linkage optimizations. This should improve system speed by about 50% on the average, and give us a system which runs between 4 and 40 times more slowly than LITTLE.

ii. To work out the data structure choice scheme fully, and to implement it. This should double, or in some cases even triple, system speed.

It seems to me best to push implementation of (i) in parallel with the development of SETL algorithms for (ii).

The reason is that much of the work involved in (i), specifically typefinding, destructive use finding, and detection of non-recursive subprocedures, will have to be done even after data structure declarations are provided, and another major part of (i), namely inclusion/membership analysis, should also be performed since it will help check the validity of a user's data structure declarations. Given these facts, it seems appropriate to let the relatively well defined parts of the optimizer move ahead to implementation even though other parts of it still await detailed definition.

Implementing the optimizations (i) will involve the following steps:

(1) Revision and cleanup of the present set of global analysis algorithms.

(2) Transcription of these algorithms into MIDL. As output, the MIDL codes should produce intermediate text in which the types of variables are made explicit, 'short' integers are noted as such, and cases in which operations can be performed destructively are noted. Dead set-valued variables should be set to Ω to facilitate garbage collection. Cases in which SETLT performs an operation destructively without the legitimacy of doing so being confirmed by global analysis should be flagged and should cause the emission of warning diagnostics. Subroutine linkage optimizations should include detection of nonrecursive routines, which can eliminate considerable amounts of stack manipulation and simplify access to temporary variables; and should also detect parameters-not-read/parameters-not-written, for which some of the ordinary parameter transmission overhead can be elided.

Inclusion and membership information gathered by the global analysis routines should be made available in the form of program annotations.

(3) The present SETL run-time library must then be extended to include routines which perform existing SRTL functions, but

perform them more efficiently by exploiting knowledge concerning types. The simplest of these routines can in fact be macros intended for in-line compilation.

(4) Routines which transform intermediate text into LITTLE code (which of course will consist largely of stack manipulation code and of calls to the extended run time library) must be coded.

The steps involved in development and subsequent implementation of the data structure choice system (ii) are as follows:

(1) Definition of a set of routines, or rather 'routine skeletons', which realize the operations of SETL for objects represented in all possible basings. Versions of these routines showing the effects of all useful elisions must also be prepared, and 'time formulae', which give the expected execution time of each routine must be developed. This material, when complete, will extend the present SETL specification of SETL considerably.

(2) The 'routine skeletons' that we have must mentioned are skeletal rather than complete because the skeleton *rskel* invoked, to handle an object o of given representation will contain calls to SETL primitives whose actual form will depend on the manner in which the subparts of o are represented. By substituting appropriate, specific routine calls for these 'generic' calls to primitives one can produce actual routines from routine skeletons. Algorithms governing these substitutions need to be worked out.

(3) When the algorithms needed for steps (1) and (2) are complete, they should be implemented. Once this is done, a mechanism which can generate indefinitely many codes that perform set-theoretic operations (on objects with different representations) will be available. Then these routines, and the whole system which produces them, will need to be debugged. Once debugged, these routines can be timed; the timing figures thus obtained will tell us how much speed gain our data structure choice system can possibly attain.

(4) A detailed syntax for data structure declarations will have to be designed and installed in the front end of the optimizing SETL system. Algorithms which check declarations for consistency with globally gathered type and inclusion/membership information will have to be built up also. Inconsistencies should be treated as fatal errors producing diagnostics; cases in which data structure declarations go beyond the global information gathered by the compiler should be flagged with warning diagnostics.

(5) To complete the system, one will need to construct algorithms which deduce the representation of every object appearing in a program, which build up appropriate routines for each SETL primitive which the program invokes, and which compile calls to these routines (or which insert them in-line where this is more appropriate).

Deficiencies of the present SETLC system

SETLC is no bulkier than SETLA, but compiles programs, especially short programs, much more slowly. To develop an optimizer the present SETL front end will have to be redesigned, and this makes any heavy interim front end reconstruction unlikely. However, a few easy patches to the present system can probably increase its compile speed significantly, and it is probably worth making these. One very easy patch is to use a version of the run-time library with the SRTL macro-decks predigested, which for short programs will save considerable time in the final (LITTLE) compile phase.