SETL Newsletter # 213

Future Plans for The
SETL Optimizer

M. Sharir
January 1,1980

The work on the SETL optimizer is now approaching the end of another phase. A first implementation, written in SETL, is about to be completed. It contains most of the major optimizations that we have contemplated, including common sub-expression elimination and code motion, type analysis, automatic data-structure selection and copy optimization. It has been tested on a few small to medium size SETL programs with satisfactory results.

Of course, in its current state, the optimizer is still far from the state at which it could be used routinely as part of the SETL compiler. Currently it does not contain based data-structure declarations, and because of its length it is compiled and executed as nine successive phases. On the average the optimizer currently processes one SETL source line per minute. However, we can estimate the speed-up that could be achieved by the following improvements:

(a) Introduction of based representations (especially in the sections which perform bitvectoring data-flow analysis, where currently the analysis bitvectors are still represented as unbased sets). This should give a speed-up of at least 2.

(b) Improvement of the SETL system, e.g. by generation of hard code and elimination of the interpretive overhead, should give a speed-up of approximately 4.

(c) Elimination of the binary I/O currently needed for communication between subsequent phases of the optimizer, and minimization of the dumps and other printouts currently produced by the optimizer, should give a speed-up of approximately 4.

All these improvements together could bring the optimizer to a level at which it could process roughly 30 lines/minute.

(d) Optimization of the optimizer by self-application. This might double the optimizer speed once more. Note however that the optimizer is roughly 10,000 lines long, so self-application will initially require a run of approximately five hours. If these estimates are correct, then the optimizer will reach only a minimal production level after these improvements.

It therefore appears likely that we will want to recode all, or significant parts of the optimizer in a lower-level language (probably LITTLE).

In any case, one of the next steps in the work on the optimizer ought to be improvement of its performance by steps (a), (b) and (c) above ((d) is too impractical at the moment.) In particular we should try to compile the optimizer as one unit. Measurement of the speed-up obtained by these improvements will allow us to be precise concerning the ultimate speed-up likely to be gained without recoding of the optimizer.

Another direction that we ought to pursue soon is extensive testing of the optimizer. In fact we ought to develop a comprehensive test-library for the optimizer. This should precede any attempt to recode the optimizer, and would require at least two to four man-months.

There are also various sections of the optimizer that have not yet been written, or that need modification. We list the most important ones:

(1) Add to the automatic data-structure selection a refinement phase which chooses local, remote or sparse attributes for based objects, in a way which is based on the usage of these objects.

(2) Consider various other extensions to automatic data-structure selection, such as list representation for tuples, multiple REPRs, etc.

(3) Add the currently missing 'bookkeeping' optimization routine to copy optimization, so that secondary improvements in the copy mechanism will be attempted,

e.g. suppression of share-bit settings and motion of copy operations out of loops. Also add special case treatment of uses of the form 'f(x) with:=y', in which local analysis can usually assist in the elimination of copy operation.

(4) Modify the treatment of partically compiled programs by the optimizer. The current approach, which simulates external procedures in the 'most general' way, seems too cumbersome, and can probably be replaced by a more compact handling of such procedures.

(5) Add a 'peephole' optimization phase in which various local code improvements are performed.

(6) Look for other major optimizations that have been overlooked in the current optimizer design, but which might become obvious e.g. by studying the Q1 code produced by the optimizer.

(7) Develop methods for applying the optimizer as an error-detecting global analyzer during the compile phase, and also to assist in management of large systems of programs.

(8) Design, and if possible install, mechanisms for the optimization of backtracking.

Beyond all of this, a significant possibility seems to be the development of 'multi-level' optimization techniques, i.e.

(i) Begin by determining appropriate data structures, eliminating unnecessary copies, etc. After this, determine those parts of the library which will be entered, and assemble the relevant library fragments into a program-specific package of online macros and tailored offline routines.

(ii) Analyse this code globally and optimize it at the LITTLE level. The resulting code should compare in an interesting way with applications code written directly in LITTLE.

Note that a more detailed 'REPR' language, which facilitates data packing and the application of other LITTLE-level techniques by declaring set size and other representation-related information, may become appropriate in connection with steps (i) and (ii).

In summary, we can say that although much still needs to be done in order to realize the full potential of the SETL optimizer, we have now come very close to the point at which most of the goals initially projected for the optimizer have been realized, and least in their essentials. This now makes possible a period of large-scale experimentation out of which new ideas will hopefully grow.

Constant Propagation
Dead Code Elimination