SETL Newsletter # 136

J.T.Schwartz
July 16, 1974

A Framework for Certain Kinds of High-Level Optimisation

The optimisation techniques used in Newsletters 130 and 131 suggest the outline of a general framework for optimisation. The present newsletter will sketch this framework. Hopefully, our crude sketch can serve as a useful guide for later more detailed work.

1. In the scheme to be described, we try to regard global optimisation processes as beginning with the formation of a set IH of hypotheses concerning a code C to be optimised, and as going on to select one or more mutually confirming subsets from among these hypotheses. The initial set of hypotheses can either be chosen in some *a priori* manner relatively independent of the details of C and then quickly pruned to eliminate implausible hypotheses, or it can be generated by some process of transitive closure which starts by examining C and collecting salient features which suggest hypotheses. The initially formed hypothesis set IH should not be so narrow as to exclude plausible facts of potential interest from consideration; neither should it be so broad as to include facts unlikely to be of reasonably common use.

2. Once an initial set IH of statements is chosen, a maximal subset MC of mutually confirming statements can be found by discarding statements which conflict with observed features of C, and then recursively descarding all statements which have a discarded hypothesis as precondition. This rule applies to what may be called *secondary* statements, i.e. statements whose truth/falsity depends entirely on the truth/falsity of other statements belonging to the same statement set.

Statements of another type, which we shall call *primary*, can also appear in our statement sets IH and MC. A primary statement represents either an internal decision to be made at compile time or a fact concerning the data objects appearing in C but not decideable by compile-time analysis. The statements 'x will be represented by a list', 'x will be represented by auxiliary bits attached to the representation of y', 'x will be represented by a hash table' are examples of primary statements of the first kind. The statements 's is a set containing few elements', 's is a subset of s_1 and a substantial part of s_1', 's_1 is a set containing many elements' are examples of primary statements of the second kind. As these examples show, primary statements will normally belong to small statement groups, the statements of a group being mutually incompatible. Such a group of primary statements represents a choice that must be made by an optimiser either autonomously or under interactive guidance.

In reducing an initial set of plausible secondary statements to a mutually confirming subset MC, we will find that certain of the statements in MC depend on hypotheses which are primary statements, and conversely that the truth/falsity of certain primary statements determines whether particular secondary statements will be admitted into MC or not. Primary statements for which this is true will be called *relevant* primary statements.

Relevant primary statements representing facts concerning the objects of C not decidable by compile-time analysis can be put, perhaps interactively, as questions to a programmer. Programmer denial of hypotheses required to justify some otherwise self-consistent system of assertions will drop certain such assertions out of consideration. If a relevant primary statement represents a decision to be made internally at compile time, the optimiser can explore all the alternatives which grow out of possible truth values for this statement.

This will generate several, hopefully not many, plausible alternative realisations of a given program; then these can either be described to a programmer who has the right of final choice of one of these approaches, or 'rated' internally in some quantitative way for the automatic choice of a 'best' approach.

3. It may be possible to force the treatment of the type of optimisation proposed by Jay Earley ('iterator inversion' or 'generalised reduction in strength') into the framework proposed in the preceeding paragraphs. This optimisation takes an expression E whose evaluation would involve extensive calculations but whose parameters are changed incrementally by some body of code, and maintains its current value V rather than recalculating V each time E is encountered. The value V is kept current by attaching appropriate updating operations to each statement which modifies a parameter of E. To force this type of optimisation into our suggested framework, one would introduce a type of symbolic statement σ with the heuristic interpration 'E is to be maintained as a current value' into the analytic framework. Statements of this kind would have secondary statements such as 'all (or some) of the parameters of E are modified differentially' as necessary preconditions; moreover, pirmary statements such as 'it is expensive to evaluate E' might also be attached to σ as necessary preconditions.

It is significant that any application of the technique of optimisation by generalised strength reduction opens the way for additional applications of the same technique, since each such application ensures that some additional expression E is only modified differentially.

This makes chains of successive optimisations possible and allows rather extensive transformation of an initially given program text. The individual transformations T in such a chain will be easy to apply, and it will be easy to determine when application of any specific T is plausible. However, to determine whether all the detailed conditions required for T's application to be fully valid may require more extensive calculations. For this reason, it might be appropriate to use the following technique. Explore chains of plausible optimising transformations rather broadly; then determine the expected efficiency of the programs which result. From this set S of programs choose the program P of greatest efficiency and only then attempt full validation of the chain of transformations leading to P. If this chain is validated, accept P as the optimal element of S. If validation fails, choose the next most efficient program in S and attempt its validation, etc.