SETL Newsletter 135C August 1, 1975

On the Genesis of Complex Programs J. T. Schwartz

Certain classes of programs are very much more complicated than the direct statements of the problems which they solve; it is interesting to ask how and why this complexity arises. Let us begin by considering optimizers; in many ways, this subclass of complex programs can typify the whole class. The problem which an optimizer O solves is: given a program P (perhaps one written in an abstract language), transform it into an equivalent but more efficient program O(P); for example, we may want O(P) to process a few typical data sets ten, or one-hundred, times as fast as P if this is possible. A skilled programmer will be able to solve this kind of problem without difficulty, if P is not too large. But naive calculation of O(P) directly from the definition of O(P) is completely out of the question in every case, since such calculation would involve a mathematically vast number of steps, e.g. might require generation of all programs provably equivalent to P, followed by numerous efficiency tests of the programs generated.

To overcome this difficulty one proceeds as follows. A collection of transformations t which send programs P into equivalent programs P' is devised. The transformations t are chosen so as to map programs P into more efficient programs P ; however, to be sure that tP is more efficient than P one may have to be sure that some associated condition $C_t'(P)$ is satisfied. The applicability of a particular transformation t may only be guaranteed if some associated boolean condition $C_t''(P)$ is satisfied. Let $C_t = C_t' \wedge C_t''$; to optimize a program P one then wants to apply an appropriate sequence $t_n \ldots t_1$ of the transformations t to it; this sequence must be chosen in such a way as to ensure that $C_{j+1}(t_j \ldots t_1 P)$ is always satisfied. It is of course not feasible to investigate all possible arrangements of all the transformations t. Instead, roughly the following scheme can be used: from P, precalculate a seuqence $t'_1 \ldots t'_m$ of transformations t such that $C_{i_1}(P) = C_{i_1}(t'_{i_2} \ldots t'_{i_k} P)$ for every subsequence $i_1 \ldots i_k$ of $1 \ldots m$. This allows the applicability of a particular transformation t'_j to be decided independently of what other transformations of the sequence $t'_1 \ldots t'_n$ are applied. Then all t'_j for which $C_j(P)$ is true can be applied to P.

The routine which precalculates $t'_1 \ldots . t'_m$ from P can appropriately be called an *organizing framework* for the optimizatin process.

The mass of transformations devised to ensure effectiveness of an optimizer process like that just sketched can and will grow unboundedly in size as optimizing transformations aimed at new, ever finer aspects of programs are developed. Moreover, as one increases the number and variety of transformations which an organizing framework must coordinate, the complexity of the framework itself will increase. This increase will be compounded by the fact that optimizer efficiency becomes steadily more important as the number of allowed transformations is increased, since then more and more conditions $C_j(P)$ need to be calculated. Since P cannot be speeded up by more than a limiting constant factor, the amount of effort one is willing to expend on these calculations is limited. Thus as more and subtler transformations are admitted to consideration, one will feel compelled to calculate their associated conditions $C_j(P)$ in more ingenious and complex ways, e.g. by devising global program functions which can be calculated efficiently, and from which the conditions $C_j(P)$ can be calculated with special rapidity. One cannot expect to proceed very far along this complicating path before reaching an absolute limit of programmability. This makes it clear that optimizer design, like mathematics, must ultimately rest on the fortuitous discovery of lucky cases; more specifically, cases in which one can define classes of particularly simple transformations having notably beneficial effects on program efficiency, especially if the applicability of these transformations can be checked with exceptional ease.

The situation that we have depicted arises in connection with programs other than optimizers. For example, error correction problems can be cast into the following form: A string P (which may be 'incorrect') is given, and one wishes to find the closest correct string to P, or, at any rate, a correct string P' which is not much further from P than the closest correct string. Here the distance between two strings can, e.g., be measured in terms of the number of symbols in which they differ, and string correctness can be defined by use of some sort of formal grammar, to which additional programmed 'semantic conditions' C(P) = true may be appended. Comprehensive exploration of the whole neighborhood of P is infeasible; instead of this, one devises a set of transformations t ('error correctors') each of which maps P into a string tP which is closer to being correct. Each of these transformations will only be applicable and appropriate if some associated condition $C_t(P)$ is satisfied; so in formal terms this situation resembles that discussed above. Another problem which can be cast into much the same form is that of algebraic manipulation.

In tackling problems of this sort, the program designer aims to reach (or surpass!) the human level of problem competence without exceeding the limits of programmability. A method useful in working toward this goal is to collect and classify the various techniques which play an important role in the treatment of some particular class of problems, and then to devise a general, programmable approach which dominates all (or at any rate most) of these techniques. This very problem-specific approach is open to an important strategic objection: it is inextensible, especially since it always tends to operate near the boundary of programmability. If there is a larger hope implicit in this approach, it is that significant simplification in many particular areas will eventually come out of separate detailed problem solutions, eventually allowing these separate solutions to be welded together into very broad multi-functional systems. By contrast, an 'artificial intelligence' approach will insist from the start on primitiveness of method; only grudgingly will complex problem-specific material be admitted. At the present time, this approach will amost always founder amidst efficiency problems. The hope implicit in it is that by devising particularly powerful generalizations, by discovering particularly fortunate problem representations, and by building very general mechanisms for the digestion of problem specific material this inefficiency can be overcome.