SETL Newsletter # 135

Introductory Lecture at the

June 28 'Informal Optimisation Symposium'

J.T. Schwartz

July 1, 1974

Most of the talks at this informal symposium on optimisation of high level languages will concern SETL optimisation. This introductory lecture aims to describe and project the optimisation effort which the SETL group is now beginning, and to make a number of general points which put our present work and planned effort into perspective.

The optimisations which can profitably be applied to any programming language L (whether this be FORTRAN, PL/I, or SETL) fall naturally into two categories. Some optimisation problems are artifacts of the language, in that they depend in a relatively sensitive way on particular details of L and would not arise had certain features of L been defined differently. Other optimisations are basic, in that they will necessarily arise in connection with any language of roughly the same level and general nature as L, whenever such a language is translated into a language of lower level. To illustrate this distinction, we note that it is the declaration-free nature of SETL which makes it necessary to analyse a SETL program to discover the types of the objects which it uses. Had type declarations been included in the specifications of SETL (which would have led to a language of somewhat different flavor, but not to an essentially different language) optimisation by typefinding would have been unnecessary. Similarly, copy optimisation is necessary in SETL because consistently value-oriented semantics have been specified for SETL (as for APL).

Had value oriented semantics been insisted on less firmly, and some of the responsibility for the insertion of object-copying operations left to the SETL user, copy optimisation would have been a much less important part of the SETL optimisation problem than it actually is. For this reason both the typefinding problem and the copy optimisation problem may be described as artifacts created by particular details of the specification of SETL. In contrast, optimisation of SETL programs P by choosing concrete data structures which efficiently represent the abstract objects appearing in P is an inherent problem bound to arise whenever a language of the level of SETL is compiled down to a language having roughly the level of PL/1.

We note that even the optimisations which we have classified as artifacts of a language's definition are worth study. By developing methods for handling optimisation problems of this type, we can hope to learn how to treat more central and fundamental problems. Moreover, contingent problems are apt to be easier than fundamental ones, since knowledge of the declarations or slight redefinitions of language which would cause these problems to disappear gives us a valuable clue concerning the global information which needs to be gathered if they are to be optimised away. For this reason, to redefine a language in order to eliminate (soluble!) optimisation problems is not necessarily a terribly worth-while undertaking. Optimisation problems will certainly arise in the treatment of any language which is not simply an assembly language; slight language redefinitions can postpone but not eliminate these problems.

At any rate, the problem of data structure choice is certainly central to the question of optimisation of programs written in a language of very high level. Such programs in general, and SETL programs in particular, can be considered to represent algorithms as they exist before the detailed data structure choices which are required for algorithm realisation in a language of lower level (such as PL/1 or ALGOL 68) have been made. It thus falls to a SETL optimiser to choose both the data structures which will represent the abstract objects of a SETL program and the code sequences which will realise the abstract operations to be performed on these objects. Ideally, these choices should be made automatically using facts collected during analysis of a SETL text to be translated. It is seen however that some parts of the information relevant to data structure choice (such as the frequences with which certain operations will be executed, or the expected size of certain data objects) are in fact not deducible by an optimiser which is given the text of a program and nothing else. Faced with this obstacle to fully automatic optimisation of SETL, one may decide full back on a semi-automatic scheme, in which an optimiser works both from the text of an algorithm and from supplementary set of hints or declarations, which we might imagine to be roughly analogous to the optional 'frequency statements' of an old version of FORTRAN. Note however that a satisfactory semi-automatic optimisation scheme is characterised by the fact that the extra information it requires supplements the text of algorithm as this exists in pure SETL, but does not imply anything more than light rewriting of an original SETL text.

If to make them efficient programs must be extensively re-written, we have a scheme for compiler-assisted manual transcription rather than a declaration-assisted optimiser.

To choose data structures and code sequences which efficiently realise more abstractly stated algorithms is at present a very central part of the programmer's work. If we are able to sterotype this process of choice and make it automatic we will have taken a major step toward the realisation of 'automatic programming'.

On the other hand, if investigation shows these choices to be highly varied and not subject to regularisation , then programming may for a long while remain as much a manual endearour as mathematics. Note that in attempting to re-gularise the process of data structure choich we do not ask whether all, or even many, of the data-structure related decisions actually made by an experienced programmer can be duplicated automatically. Rather, we ask whether we can find some rather narrow subfamily of the family of all possible data structure choices, doing this in a way which guarantees that some choice in our subfamily is an adequate, replace-ment for any choice which a programmer is likely to make.. If this can be done, we can regard the remaining devices used by programmers as irrelevantly personalistic variations which complicate the process of coding without really improving it. The situation which we anticipate may be compared to that encountered in translating a language of the FORTRAN level down to machine level. An assembly language programmer will assign registers to variables in highly varied ways; but the quite sterotyped action of a register allocator is seen to produce code which is just as good, and this allows us to consider much of a machine level programmer's activity as a process of personalistic, and hence ultimately undesirable, variation.

Concerning the processes which enter into data structure choice we are still uncertain. What information does a programmer use in choosing data structures, and how does he use it? Although some of the talks to be given later today will begin to illuminate this question, a large part of our answer must still be-we do not know. How then can this question be approached? One important initial line of approach is to make a crudely empirical case-study of the issues arising in data structure choice. For this, one requires a source language of high level (in our work we use SETL), and a target language of lower level into which it is to be transcribed (for this purpose, we shall probably be using a language of a vaguely PL/1-like semantic level, but one which provides a garbage-collected memory millieu; this language has provisionally been designated as GLITTLE.) Some more specifically applications-oriented language than SETL might be used as source language in such an endeavour; however, SETL can claim as an advantage both the fact that it is quite general and the fact that it is a language directly appropriate for writing the very optimisation algorithms which one aims to develop.

Once having chosen suitable source and target languages, we pursue our empirical study by taking a representative variety of algorithms written in the source language and translating them manually into equivalent but efficient algorithms in the target language. We aim to do this as systematically as we can, and in a highly 'self-conscious' way. It is a good idea in translating a program P to work from an explicit list of the objects appearing in the source program, and to note carefully all facts concerning the nature of each such object O, the way in which it is used, and the relationships which O may bear to other objects appearing in P, which are significant in selecting O's representation at the lower lever of language.

It is also well to aim at a transcription of P which, while efficient, is as close to P in abstract structure as is possible. By noting the facts which repeatedly appear relevant when this process is applied to program P, we take a first essential step toward mechanising data-structure choice. In FORTRAN, a similar step was taken with the observation that in doing a careful allocation of registers a programmer will make use of everything he knows about the live/dead status of variables; this was the crucial observation which made optimised use of registers possible.

Deeper initial observations must be made in order to get well started with the more complex problem of automatic data structure choice. A preliminary study of examples highlights some of the factors governing such choice. To know the type of each of the objects appearing in a SETL program P is important. It is also important to know any relationships of inclusion and membership which can be shown to persist throuthout the execution of P. We will also want to know the pattern in which operators op are applied to objects, and, if necessary, to trace this pattern of operator-to-object application through all chains leading from the initial creation of x, through any statements inserting x as an element or component into a set or tuple, through later extractions of x from such a set or tuple, up to the point at which the operator op is ultimately applied to the body of x. As such information becomes available to us we come into position to choose special representations for x. Note for example that if x is known both to be a set and to be a subset of some other set y appearing in the same program P, then we do not need to maintain the standard SETL representation of x, but can simply associate one extra bit with each of the elements of y, and use this bit to indicate whether a given element of y also belongs to x.

This technique can be used directly if x is never made an element of a still more compound object; if,on the other hand,x is made part of a compound object c, then the bits which flag elements of x must be collected into a bit-vector which can be inserted into c in lieu of x. If x is used only in membership tests and to form unions and intersections, then a pure bit-vector representation of x may be adequate. However, if iterations over x are performed, and if x is neither a very small set nor a very large part of y, we may wish to use both a bit-vector and a list of the elements of x to represent x.

Although fragmentary, the preceeding reflections do begin to indicate the way in which the choice of representing structures grows out of a programmer's knowledge of facts concerning the various objects which appear in an abstract algorithm. Once we have defined the class of facts which will enter into the data structure choices which we hope to make automatically, our problem becomes that of building up global program analysis algorithms capable of establishing these facts. We offer the hypothesis that when we know what information is wanted, it will not be terribly hard to devise algorithms capable of collecting this information. Indeed, the material to be presented later today reveals the first outlines of an analytic approach. We also surmise that carrying out a full measure of automatic analysis will remain necessary even if one aims to build not a fully automatic, but only a semi-automatic or an interactive optimisation system. Only after extensive program analysis has narrowed an initially very large family of possibilities down to a set of two or three crucial choices can as optimiser system either accept hints stated at a reasonable dictional level or emit sensible questions.

We therefore see a semi-automatic or an interactive optimiser as a automatic optimiser which relies on its user to supply a few final facts, and not as a very different, easier to program kind of system.

It is worth making a few more specific remarks concerning the analytic parts of an optimiser system. These routines are in effect specialised theorem provers which prove facts about programs. But, in contradistinction to some of the other types of program-related theorem provers which have been considered in the literature, they operate in a 'high density' rather than a 'low density' range, i.e. they prove numerous small and easy facts concerning programs P (such as the fact that the values of one set valued variable s_1 are subsets of the values some other s_2) rather than proving one or two big, hard facts concerning P (such as correctness and termination). For this reason, it may be better to call optimiser-associated program analysis routines *fact gatherers* rather than *theorem provers*. We can put this comparison somewhat differently by considering the technical nature of the theorem provers which are employed by program analyser/optimisers on the one hand, and by program-correctness verifiers on the other. Theorem proving programs fall into two main families: on the one hand, those generically similar to the original 'geometry theorem prover' of Gelernter; on the other hand, those belonging to the resolution group. Provers of the first kind proceed very cautiously in generating objects not almost explicit in the situations with which they are presented. This limits very significantly the space of possibilities which such a prover needs to explore, and makes it possible for such provers to generate facts using what is essentially a transitive closure method.

Provers of the second kind are more general, and in principle capable of reaching out much further from an initially given set of hypotheses, largely becuse they have available, and are prepared to use, constructor mechanisms capable of generating all the objects of some full 'Herbrand universe'. However, their very generality confronts provers of the second type with the problem of searching rapidly growing, potentially infinite sets of possibilities, and at the present time provers of this second type generally founder amidst multitudes of unexplored possibilities.

Fact-gathering analysis routines associated with program optimisers can be expected to use the limited method of 'proof by transitive closure' rather than the more general 'resolution' method. This observation is certainly valid for all the program analysis routines which will be described in the talks to follow. These routines have another noteworthy characteristic in common. Each is built around some 'algebra' A of properties or relationships upon which SETL programs P act symbolically in a manner homomorphic to the detailed action of P on its environment during actual execution. All these algebras are finite enough for the symbolic action of P on A to stabilise after finitely many steps, which implies that the interaction of P with A is a matter which can be fully worked out at compile time. In implementation terms, such algebras A are represented by medium to large tables whose seperate entries describe the action of each of the primitives of the language L to be analysed (in our case SETL) on the symbolic entities of A. Such a table defines the basic 'knowledge' concerning L which an analysis algorithm will have; a suitably structured process of transitive closure, almost common to all our analyses, distributes this knowledge in a suitably global way over a program to be analysed.

These reflections emphasise the very large part which an understanding of what we are looking for is apt to play in our total approach to the problem of program analysis.

It is worth observing that a similar approach, i.e. analysis of a program P by symbolic compile-time application of P to the elements of an associated algebra, emerges in a recent IBM technical report (Yorktown Research) by Gernot Urschler.

Certain of the most basic algebras A used in global program analysis, as for example the Boolean algebra of bitstrings used to determine operation redundancy, basic data-flow relationships, and variable live/dead status, have special properties which allow the action of P on A to be calculated in just a few iterations. For certain of the other algebras A to be described later today no such principle is available, so in working out the action of P on these A we are forced to use algorithms which simply iterate over the control or data flow of P until our analysis stabilises. Perhaps it would be better to say that, knowing no better algorithm at the present time, we use a crudely iterative technique in our analysis. This latter formulation emphasises the fact that in the present primitive state of our understanding of many of the optimisation processes to be described today, we are concerned more with the specification of *some* algorithm capable of deducing important program-related facts than with the choice of efficient fact-gathering algorithms. However, it is to be expected that improvements of our algorithms will follow rapidly upon their initial statement.

We have emphasised that optimiser-associated programs analysis routines are bound to search for numerous small and easy facts concerning programs rather than for deeper but much less easily established facts. Indeed, the difficulty which an automatic theorem prover experiences rises with immense rapidity as the depth of the problems presented to it increases; hence in optimising it is only profitable to search for facts which can be routinely established. Generalising, we can assert that at the present time automatic theorem provers are only likely to apply sucessfully to problems which the mathematician can regard as routinely soluable, i.e., problems for which general approaches are known and for which a satisfactory approach can be deduced in a reasonably straightforward way from the problem itself. It is worth noting that programming itself has this same character: programming begins with what mathematics considers to be a problem's solution (i.e. with an algorithm formulated in general outline), and the thought processes which programming involves have (at their best) the character of systematic elaboration rather than of discovery. The characteristic difficulties of programming arise from the fact that the programmer is forced to work for extended periods at complexity levels close to the maximum threshold of sustainable complexity, which inevitably introduces errors into his product, errors whose removal is a very large part of his actual work. From this definition of the process of programming we conclude that it should be subject in large part to automation. More specifically, it should eventually be possible to build systems which accept abstract formal process definitions (written at roughly the SETL level, or at a level somewhat higher) as input, and which are themselves responsible for the routine but high-complexity steps of the programming process. We expect however that a system of this kind will be capable of making only routine but not deep deductions, so that the input text presented to such a system will have to describe every mathematically essential aspect of each program which the system is to develop.

Similar considerations apply if we attempt to define the notion 'programming language'. If L and L' are notational systems and if L is more abstract than L' but can be translated into L by a process which makes use only of routine deductions, then we may say that L and L' are related as higher and lower levels of programming language. On the other hand, if reduction of L to L' is mathematically possible but requires the deduction of deep facts, then L may be a useful mathematical system but is not really a programming language compilable into L. By sophisticating his source language to a point at which it can only proceed by making deep deductions, the designer of a system for automatic programming can easily bring himself to shipwreck.

Deep, hard-to-prove logical facts are publishable mathematical theorems. If we accept the assertion that automatic theorem provers will for the present only be able to prove much more superficial results, we must ask the question: in what applications are we likely to find use for masses of specialised and relatively superficial facts? Various possibilities suggest themselves. Optimising translators between levels L and L' of language require such facts. Data base systems may eventually incorporate dynamic algorithms which minimise the size of the search generated in response to a quary, and dynamic optimisers of this type may come to have fact-gatherers as components. Data bases of certain types can probably be compressed by storing only some of the more fundamental facts relevant to a given area and leaving others to be obtained by deduction; and routinely deduced information would clearly be useful in such a system. Finally, we note that there are a few situations in which the outcome of a large set of routine but tedious deductions can be directly useful to a person. Interactive algebraic manipulators are typical of such applications.

The fact-gathering processes within an optimiser are inescapably global in character. The data which these processes collect must ultimately be cast into some suitably localised form, since these facts must ultimately be used by a code generator which we expect to act in 'peephole' fashion. However, data structure choices have global implications, and must be made coherently for the whole of a program. Thus only after global data structure choices have been made can we expect code generation to become a problem treatable locally. Exactly how to treat the global interactions which will confront us in making these choices is a new and nontrivial problem, and not necessarily one which can be wholly absorbed into the design of the fact-gathering process which preceeds data structure choice.

Many of the algorithmic investigations to be reported on later today have the development of a comprehensive SETL optimiser system as their ultimate goal. At present, we expect this goal to be reached via the following sequence of steps. First we must complete the work whose earliest phases will be reported on today: the specification of numerous seperate optimisation algorithms, essentially one for each major class of program-related facts to be gathered. Next these seperate algorithms must be integrated into a design for a comprehensive SETL analysis system; to prepare this system for implementation, we must write it out in SETL. At this point, implementation proper will begin.

The next step of actual implementation will be the development of a program to translate SETL source code into some appropriate intermediate text, probably resembling the 'quadruples' used as input to A. Tenenbaum's typefinder. This translator can be obtained by modifying the present SETL parser. Once we have a source of intermediate text we will be able to debug the SETL text of the comprehensive analysis algorithm, thus making a SETL analyser available in a first running version. By attaching a relatively simple back end to this analyser, we will be able to use it as a program annotator; used in this way, it will simply attach the facts which it gathers to the SETL texts presented to it for analysis. By running a variety of texts through this annotator, we will be able to assess the completeness with which it uncovers potentially available facts, and to modify it to achieve greater completeness if necessary. Once a relative exhaustive fact-gatherer is in hand, a detailed data-structure choice algorithm can be designed and developed. Our last design task will be the definition of a peephole optimiser capable of using the results of all the preceeding analyses to generate good LITTLE code. Finally, all the algorithms of the rather extensive collection which has just been sketched will have to be realised in production versions, probably using GLITTLE. All this clearly adds up to a major undertaking, but hopefully one that we can carry through, and hopefully one that by doubling or tripling the efficiency of SETL will move it into a performance range in which it can be of wide appeal to a substantial body of users.