A laboratory for the study of automating programming*

by T. E. CHEATHAM, JR. and BEN WEGBREIT

Harvard University
Cambridge, Massachusetts

INTRODUCTION

We are concerned in this paper with facilities, tools, and techniques for automating programming and thus we had best commence with discussing what we mean by *programming*. Given a precise specification of some task to be accomplished or some abstract object to be constructed, *programming* is the activity of producing an algorithm or procedure—a program—capable of performing the task or constructing a representation of the object on some computing system. The initial specifications and the resulting program are both couched in some (programming) language—perhaps the same language. The process typically involves such activities as: choosing efficient representations for data and algorithms, taking advantage of known or deduced constraints on data and algorithms to permit more efficient computations, verifying (proving) that the task will be accomplished or that the object constructed is, in fact, the one desired, demonstrating that certain performance criteria are met, and so on.

The kind of facility currently available which might be characterized as contributing to automating programming is usually called a *compiler*. It typically translates an algorithm from some higher level (programming) language to a lower level ("machine") language, attempting to utilize memory and instruction resources effectively and, perhaps, reorganizing the computational steps, as implied by the higher level language representation, to move invariant computations out of loops, check most likely (or cheapest) arms of conditions first, and so on.

We are not here concerned with traditional compilers; indeed, we will assume the existence of a good compiler. We are concerned with facilities at a "higher level": translating specifications which contain much less commitment to particular data and algorithmic representations than is usual with higher level programming languages, and performing rather more drastic reorganization of representation, implied computational steps, and even implied method of computation than is done with traditional compilers. We imagine our end product to be programs in a higher level language. On the other hand, we must note that the line between the kind of facility we will describe and a good compiler is very fine indeed and we will suggest that certain kinds of transformations sometimes made by conventional compilers might better be approached with the tools and techniques described here.

The purpose of this paper is to describe a facility which we characterize as a laboratory for the study of automating programming. We view the laboratory as a pilot model of a facility for practical production programming. The time and expense invested in programming today and the lack of confidence that most programs today actually do what they are intended to do in all cases is surely dramatic evidence of the value of such a facility. The need is particularly acute when the task to be accomplished is complex and the resulting program is necessarily large. Such situations are precisely those encountered in many research areas of computer science as well as in many production systems software projects. Dealing with this kind of complexity, which is to say producing efficient verifiably correct program systems satisfying complex requirements is a significant, decidedly non-trivial problem.

The second section of this paper contains a critical discussion of a wide variety of work and research areas which are related; the third section is devoted to a broad general description of the laboratory; the fourth section then briefly describes the ECL programming system, the intended host for the laboratory; the fifth

* This work was supported in part by the Advanced Research Projects Agency under contract F-19628-68-C-0379 and by the U.S. Air Force Electronics Systems Division under contract F19628-71-6-0173.

section discusses, in general terms, a variety of program automation techniques to be employed; the sixth section describes the basic components of the initial laboratory; and the seventh section summarizes what we hope to accomplish with the laboratory and mentions several open problems.

RELATED WORK

There is a considerable body of work and a number of current research areas which are related to programming automation. Most of this work does not, at present, provide anything like a complete system; much of it does provide *components* of a system for automating programming and is thus directly related to and sometimes directly usable in the laboratory we will describe.

We have divided the work to be discussed into seven different areas: automatic program synthesis, mechanical theorem proving, automatic program verification, program proof techniques, higher level programming languages, equivalence of program schemata, and system measurement techniques. In each case we are discussing the work of several people; the bibliography cites the recent work we feel is most relevant.

Automatic program synthesis

The basic idea here is to construct a program to produce certain specified outputs from some specified inputs, given predicates asserted true of the inputs and the outputs as related to the inputs. The basic technique is to (mechanically) prove the theorem that there exist outputs satisfying the predicate and then to extract a program from the proof for constructing the outputs. It has been suggested that these techniques can also be utilized to transform programs, for example to transform a recursive procedure into an equivalent iterative procedure using the two stage process of first deducing a predicate that characterizes the recursive procedure and then synthesizing an equivalent iterative procedure which computes the outputs satisfying the predicate deduced.

We view the work in this area to date as primarily of theoretical interest and contributing to better mechanical theorem proving and proof analysis techniques. It is often more convenient to produce an (inefficient) algorithm than it is to produce a predicate; the two stage process proposed for "improvement" of programs is awkward and, we believe, highly inefficient as compared with the direct transformation techniques to be discussed below.

Mechanical theorem proving

The heart of any system for automating programming will be a facility for mechanical theorem proving. At the present time there are two basically different approaches to mechanical theorem proving and a realization of both these approaches provide important components of our laboratory. One approach is to construct a theorem prover which will, given enough resources, prove any true theorem in the first order predicate calculus with uninterpreted constants; the other approach is to provide a theorem prover which is subject to considerable control (i.e., allows one to employ heuristics to control the strategy of proof) and which utilizes interpretations of the constants wherever possible for efficiency. Mechanical theorem provers of the first sort are now usually based on the resolution principle. We term those of the second sort "programmable theorem provers".

Resolution theorem provers

Robinson's 1965 paper introducing the resolution principle has been followed by vigorous activity in implementing mechanical theorem provers based on this principle. Much of the activity has been concerned with developing strategies for ordering consideration of resolvents; at the present time the breadth-first, unit-preference, and set-of-support general strategies have been studied and other variations are being considered. It is clear that a powerful resolution-principle based theorem prover will be an important component of the laboratory.

Programmable theorem provers

In the PLANNER system, Hewitt provides a facility for programmable theorem proving in the sense that one can very easily utilize interpretations of the objects and operations entering into a theorem to control the strategy of proof, one can make the choice of technique used in any particular instance data dependent, and can very readily employ any general mechanical theorem prover, effectively as a subroutine. The use of a small subset of Hewitt's proposed facilities by Winograd (called micro-planner by him; see [Winograd 71], [Sussman 70]) in his program for understanding natural language gives dramatic evidence of the effectiveness of the approach. We thus include a programmable theorem prover on the style of Hewitt's and Winograd's as the basis for the theorem proving component of our laboratory.

Automatic program verification

The work in this area is concerned with mechanization of what we term "flow chart induction". Given a representation of some algorithm as a flow chart with assignments, conditional branching, and looping, one appends to the boxes of the flow chart predicates asserted true of the variables at various points in the computation and, in particular, of the inputs and the outputs. The procedure is then to attempt to mechanically demonstrate that the whole is consistent and thus that the program is correct.

Again, we view this work as primarily of theoretical interest. The theorem proving techniques utilized in King's system (see [King]) are particularly interesting, however, as they utilize interpretations of the integers and operations over the integers; while not general, they do provide rather more efficient methods for proofs concerning integers than is presently possible with the more general resolution-type proof methods which do not employ interpretations.

Program proof techniques

A number of workers have been concerned with developing proof techniques which are adapted to obtaining proofs of various properties of programs. These include some new induction methods—structural induction and flow chart induction—simulation techniques, and so on. This work provides a very important basis for proving the equivalence of various programs.

Higher Level Programming Languages

A considerable amount of work in the development of higher level programming languages has been concerned with providing languages which are particularly appropriate for certain application areas in the sense that they free the programmer from having to concern himself with the kind of detail which is not relevant to the application area in which he works. For example, APL permits a programmer to deal with arrays and array operations and relieves him of concern with the details of allocation, accessing, and indexing of the array elements. SNOBOL 4 directly supports algorithms which require back tracking, providing the mechanics automatically and permitting one to write theorem-proving, non-deterministic parsing, and such like algorithms very easily. SETL provides general finite sets as basic data objects plus the usual array of mathematical operations and predicates on sets; it thus permits one to utilize quite succinct statements of a wide variety of mathematical algorithms and to considerably ease the problem of proving that the algorithms have certain properties. ECL (which we discuss in some detail in a later section) provides a complete programming system with facilities which permit one to construct extended language facilities such as those provided in APL, SNOBOL 4, and SETL, and to carefully provide for efficient data representation and machine algorithms to host these extended language facilities.

Equivalence of program schemata

There has been considerable interest recently in studying various program schemata and investigating their relative power, developing techniques for proving their equivalence, and so on. Most of the work to date is interpretation independent and while, for example, many transformations from recursive specification to iterative specification of algorithms have been developed, it is clear that many practical transformations cannot be done without employing interpretations.

The most common use of interpretation dependent transformations is in "highly optimizing" compilers. There, a very specific, usually *ad hoc* set of transformations is employed to gain efficiency. Often the transformations are *too ad hoc*—under certain conditions they do not preserve functionality (i.e., don't work correctly).

System performance measurement techniques

It is a quite straightforward matter to arrange for various probes, monitors, and the like to permit measurements of the performance of programs, presuming that appropriate test input data sets are available, and a considerable amount of work has been done in this area. However, there are two further areas which are now the subject of investigation which we feel may yield important components for our laboratory. These are mathematical analysis of algorithms and automatic generation of probability distributions for interesting system parameters.

Mathematical analysis of algorithms

Several people have been working recently in the area of developing methodology and techniques for the mathematical analysis of algorithms to obtain estimates and/or bounds on certain critical parameters and for developing (and proving) optimal algorithms for certain functions. We envision the manipulation facilities of the laboratory as being readily adaptable to providing

mechanical assistance in this activity, particularly in the area of aiding in the inevitable symbolic algebraic manipulation required in carrying out a mathematical analysis.

Automatic synthesis of probability distributions

Some recent work by Nemeth (see [Nemeth]) may, when it is developed further, provide an interesting and valuable component of the laboratory. What he is trying to do is to develop algorithms for mechanically generating probability distributions for various parameters from a computational schema augmented by given probability distributions for input variables and functions employed. Use of techniques like his should prove far superior to actually carrying out the computation for sample data values. A mixture of mechanical generation of distributions and carrying out portions of a computation might, in the earlier stages, provide a practical tool.

All the above work is related and relevant to automating programming, but none, in our opinion, is adequate alone. The need now is to integrate these facilities, techniques, and so on into a *system*—a laboratory for the study of automating programming.

THE APPROACH TO BE TAKEN IN THE LABORATORY

The goal of such a laboratory is a practical, running system that will be a significant aid to the construction of real-world programs. Automating programming entails transferring to the computer those facets of programming which are not carried out efficiently by humans. It is our contention that the activity most in need of such transfer is the optimization (in a very broad sense of the word) of programs. The orientation of the laboratory and the principal task to which it will be put is that of taking an existing program and improving upon it.

That optimization is, indeed, a key problem requires little defense. If "program" is taken in a sufficiently broad sense, it is easy to produce *some* algorithm which performs any stated task. Given just the right language, program synthesis is seldom a significant issue. For many problems, the most natural task description is precisely a program in an appropriate notation. The use of an extensible language makes straightforward the definition of such notation. For other problems, it may be that a predicate to be satisfied is a better task statement, but this too is in some sense a program. The line between procedural and non-procedural languages is fuzzy at best and it may be erased entirely by the use of theorem-proving techniques to transform predicates into programs (and conversely).

As we see the problem, the issue is not arriving at a program, but arriving at a good one. In most cases, programs obtained from theorem provers applied to predicates, from a rough-cut program written as a task description, or even from the hands of a good programmer leave much to be desired. Often, the initial program is several orders of magnitude away from desired or even acceptable behavior. The larger the program, the more likely this is to be the case. The reasons are generally such defects as inefficient representation of data, failure to exploit possible constraints, use of inefficient or inappropriate control structures, redundant or partially redundant computations, inefficient search strategies, and failure to exploit features of the intended host environment. Recognizing the occurrence of such defects and remedying them is the primary goal of the laboratory.

ECL AS A BASIS FOR THE LABORATORY

The ECL programming system and the EL1 language have been designed to allow rapid construction of large complex programs, perhaps followed by modification and contraction of the programs to gain efficiency. The facilities of ECL permit one to compose, execute, compile and debug programs interactively. The EL1 language is an extensible language with facilities for extension on three axes: syntax, data, and operations.

The EL1 language plays four roles in the laboratory: (1) it is the language used to construct the various components of the system; (2) it and its extensions are the language used to state algorithms which are to be manipulated by the system; (3) it is the target language for transformations (i.e., EL1 programs are transformed into better EL1 programs); and (4) it is the host language for the theorems constituting the data base.*

The features of EL1 and its host system, ECL, which are particularly relevant to the laboratory are the following:

(a) Data types (called "modes" in EL1) can be programmer defined using several basic data types (e.g., integers, reals, characters, etc.) and, recursively, several mode valued functions (e.g., construction of homogeneous sequences, non-homogeneous sequences, pointers, etc.).

* That is, the parts of a theorem (i.e., the conditions, antecedent, consequent, and recommendation list as described in the following section) are couched in an extension of EL1 and "glued together" as an EL1 procedure by operators defined as extensions; of course, the theorems are not "executed" in the usual sense.

(b) Procedures can be *generic* in the sense that a given procedure (say that for +) can have a number of different bodies or meanings and the selection of a particular body or meaning to be used is determined by the mode(s) of the argument(s) utilized in some call of the procedure. Thus, for example, it is particularly straightforward to accommodate refinements in representation of some general data type (e.g., sets) without doing violence to algorithms defined on it by associating a new mode with a refinement and defining new versions of operators particularized to that mode via the generic mechanism.

(c) The compile-time, load-time, run-time, and the like kinds of restrictions employed in most systems are not present in ECL. In particular, the ECL compiler is a program which can be called at any time; it is given a program to be compiled and a list of variables (free in that program) whose values are to be taken as fixed. It then converts the program to a form which takes advantage of whatever efficiencies it can from the freezing of values. There is no distinction between compiled code and non-compiled (interpretive) code insofar as their discernible effect when executed.

(d) ECL provides storage allocation and reclamation mechanisms which are quite sensitive to space/time efficiencies. Thus, the so-called "data compiler" goes to some lengths to utilize memory efficiently when allocating a component of a complex data structure containing several "pieces".
The use of both "stack" and "heap" mechanisms for allocation and freeing of space is also provided.

(e) ECL provides for multiple concurrent paths and permits complete user control over the environment for each concurrent path. In addition to permitting strategies such as employing, say, a general resolution theorem prover to be applied to some difficult theorem in parallel with other undertakings, this feature makes it particularly straightforward to set up and run some program in an environment of ones choosing; for example, to gather statistics on its behavior in some simulated "real" environment.

(f) Error conditions (both system detected and those defined by user programs) are generally handled by setting interrupts and each interrupt is, optionally, handled by a user defined procedure. This feature is very helpful in test execution and evaluation of subject programs and permits construction of elaborate control mechanisms when appropriate, as might be the case in controlling the behavior of the programmable theorem prover.

There are two extensions of ECL which provide particularly convenient linguistic facilities for stating algorithms; these are an extension which permits one to deal with sets and set operations* and an extension which hosts non-deterministic programs.

OPERATION OF THE INITIAL LABORATORY

The laboratory, like ECL, is intended for interactive use. A programmer approaches it with a problem specification and a set of requirements on how the program is to perform. He and the system together undertake to produce a program which meets the specifications and requirements. The intention is that the laboratory be a practical tool for everyday use, i.e., that hard, real-world problems with realistic performance requirements be brought to and handled by the laboratory.

The problem specification may be an existing program written in EL1, possibly a long-standing production program. In this case, the presumption is that its performance falls short of that required and that the concern is with automating its tuning. Alternatively, the specification may be an EL1 program written in a very liberal extension and constructed solely as a concise algorithmic statement of the problem task. There may be little expectation that such a program will meet any non-trivial performance requirements. Significant improvements may be needed even to reach the desired order of magnitude. Finally, the problem specification may be a set of predicates to be satisfied. Here the laboratory begins by constructing an initial EL1 program using as its target an extension set designed for this purpose. Again, such a program may be several orders of magnitude removed from acceptable performance.

In very general terms, the laboratory is a man-machine system for transforming the initial program to an equivalent one which meets the stated requirements. The heart of the system is a set of transformations, actually theorems concerning the language, which preserve functionality while improving the program. Deciding whether an arbitrary transformation either preserves functionality or improves the program is, of course, impossible, but decision procedures for the gen-

* This extension permits us to capture most of the facilities proposed for the language SETL. See [Schwartz 70] for a particularly cogent argument for providing sets and set operations in a higher-level programming language.

eral case are not needed here. The laboratory will employ specific transformations which under appropriate circumstances—i.e., when their enabling predicates hold—maintain program equivalence. Constructing these transformations and verifying that the validity of the enabling predicates do insure functionality will be a task assigned to humans who may, of course utilize the facilities of the laboratory to prove the validity. While the *functionality* of the transformations may be assured, such is not the case for their *effectiveness*. To obtain a sufficiently powerful set of transformations it is necessary to include many whose utility is conditional, e.g., those which are effective only under circumstances which are difficult or impossible to verify analytically, or those which optimize performance in one metric at the (perhaps unacceptable) expense of another. In general, the transformation set will include transformations which are mutually exclusive (i.e., only one of some subset can be applied) and some which are inverses (i.e., applying two or more repeatedly will lead to a loop). Hence, choice of which transformations to apply is governed specifically by the performance requirements demanded of the program and the disparity between these and the program at each state of optimization.

Determining program performance is a crucial issue. There are two basic approaches, both of which are used in the laboratory. The first is *analytic*. The system derives closed-form expressions for program behavior based entirely on a static inspection of program structure and interpretation of the program operations. Then given a description of an input data set, e.g., as a set of probability distributions for possible input values, the system can describe the exact program behavior. Whenever such closed form expressions can be obtained, this is clearly the best certification of program performance. However, at present our analytical techniques are too weak for any but the simplest programs. The second approach is that of actually running the program on benchmark data sets, data sets provided by the programmer as part of his performance specifications. Between these two extremes lies the spectrum of simulation: those portions of the program which can be treated analytically are replaced by simulation blocks and the rest of the program is run as is. The large area of mixed strategy is particularly powerful since it allows one to use only partially representative benchmark data sets yet extrapolate meaningful results from them by the use of analytical techniques.

The utility of the laboratory will be governed principally by the specificity of admissible performance specifications and the degree to which they can be met on the intended machine. Performance specifications include the obvious bounds on execution time and space. Alternatively, they might be cast in the form: as fast or as small as possible. This is, however, only a rough cut. Few problems and fewer host machines are entirely homogeneous. In real time situations, optimizing total execution time may be far less important then attaining a certain minimum for particular sections. Similarly, the total space occupied by a program and its data is far less important than the distribution of this space over several storage devices of various capacities and access speeds. Also, the intended host machine may be a multiprocessor or provide multiprocessing capabilities by means of special processors (e.g., graphics) or remote processors (e.g., a network). Partitioning the computation among the various processors so that the computation load on each is beneath prescribed limits is another task of the laboratory.

The possible transformations for obtaining the desired performance vary considerably in scope, power, and effectiveness. A sketch of those which currently seem to have the greatest payoff may give the flavor of what may be accomplished.

The most straightforward are those for reducing the space occupied by data. Any field containing literal data restricted to N possible values requires, of course, no more than $[\log_2 N]$ bits. What may not be quite so obvious is that with an appropriate programming language, simply changing declarations is all that is required to control the storage allocated, perform the coding and uncoding on data access, and handle the necessary conversions. EL1 is such a language; hence, the class of transformations is readily obtained. In the case of sequences of literal data fields (e.g., character strings), further compression can be obtained by block encoding. Relations can be represented in a variety of ways and changing from one to another often results in significant economics. Sparcely populated arrays can be changed to lists or hash tables in which the array indices are retrieval keys. Conversely, the space occupied by pointers (e.g., in list structure) can be reduced if linked lists are replaced by arrays in which relations are represented by small integer array indices occupying only a few bits.

One candidate for optimization of both time and space is sets and set operations. There are a number of particularly efficient representations for sets applicable only under certain conditions (e.g., bit vectors when the number of elements is fixed and relatively small or lists in canonical set order when the generation of new sets is carefully controlled) or efficient only when the set operations are circumscribed (e.g., hash tables when the operations are union and intersection but not set complement). When such a representation is possible, its use will often produce dramatic improvement over standard list structure techniques.

Transformations for optimizing time are often subtle and require sophisticated techniques for manipulating program structures. Perhaps the best understood sort is the transformation of recursive to iterative programs. Even restricting attention to uninterpreted schemas, there are several interesting schema classes for which the transformation can always be carried out. By adjoining additional transformations which exploit the properties of specific common program operations, a very powerful tool for eliminating program recursion may be obtained.

Time can always be saved at the expense of space by substituting the definition of a routine for a call on it. Where recursion is absent or has been previously removed, it is possible to perform repeated back substitution until all calls have been eliminated. While too costly in space to be employed everywhere, it is very effective if performed on the most frequently executed portions of the program. Aside from the obvious virtue of eliminating the expense of function call, it has the more significant virtue of allowing each back substituted defining instance to be optimized independently—in the context of the text into which it is placed.

The principal class of time optimizations is the elimination of searches. A few such transformations are simple, e.g., the replacement of association lists by hashed structures or balanced trees, and the substitution of arrays for lists which are frequently accessed by indexing. Searching is, however, not confined to lists. Structures of all sorts—arrays, queues, strings, and so on—are frequently searched for an element or set of elements having some property. When the density of hits is small, blind search is inefficient. An appropriate method is to adjoin bookkeeping records to the given structure and add bookkeeping code to the program so as to keep track of what would be the result of searching. When the incremental costs of maintaining the necessary records is small compared to the search cost, this often provides a significant optimization. Determining what records must be kept and how to keep them are non-trivial problems, but ones which appear open to solution at least for many significant program classes.

A related class of program optimizations is based on reordering computations in operations on compound structures. Any operation on a compound object must be defined in terms of primitive operations or its primitive components. Given a sequence of operations on compound objects, it is usually possible to reorder the underlying sequence of primitive operations to reduce some computational resources. Galler and Perlis (see [Galler 1970]) discuss in detail the problem of saving the storage required for temporaries in matrix operations and mention that a variation on their technique can be used to minimize time. It appears possible to generalize these results to arbitrary data structures. First, recursion is eliminated from function definitions. Then each compound operator is replaced by its definition until only primitive operators appear (i.e., the back substitution optimization mentioned above). Then, program loops are merged to carry as many operations as possible on each loop. Finally, dependency analysis is used to find common sub-expressions and eliminated unnecessary temporaries. Any number of *ad hoc*, type dependent, transformations can be added to this basic framework. The basic technique, that of unwinding compound operations and then winding them up again in a more optional fashion, is broadly applicable.

Several sets of transformations are concerned with effective utilization of the particular host machine(s). These are therefore specific to the environment, but no less important than their general cousins. The most important is that of partitioning the computation among several processors. In so doing, the first step is to take a conventional sequential program and transform it to a representation which exhibits all the potential parallelism so that sequencing is dictated only by data dependency. The next step is to partition the transformed program among the available processors in such fashion that (1) upper bounds on computational resources demanded of each machine are obeyed; (2) communication between processors is carried out satisfactorily along the available data paths and (3) the entire configuration has the desired performance characteristics. Item (2) can be reduced to item (1) by treating each data path as a processor with its own (perhaps small) computational bounds. Item (1) is, of course, the heart of the matter. The work of Holt, Saint, and Shapiro provides a very promising approach to this and has already demonstrated some success in certain restricted applications (see [Shapiro 69]).

This set of program transformations is only representative. Others will be added in time as users of the laboratory gain practical experience with and understanding of program transformation. However large such a collection, it is only a beginning and its exact composition is only a secondary issue. More important is the problem of determining which transformations to use and where, given a program and a set of performance specifications. The first step is to refine the performance measurements so as to determine precisely where the specifications are not being met. Correlating the various measurements with program text is straightforward.

Given the program text augmented with resource utilization statistics, the next task of the laboratory is to find places in need of optimization, find one or more

appropriate transformations, verify whether they are applicable and apply them. In choosing places to concentrate attention the laboratory starts simply by looking for the areas with the largest cost relative to that desired. Given these obvious starting places, the key problem is tracing back from these when necessary to the causes, perhaps at some point far removed in the program. In this step, and in the choice of transformation classes to be attempted, there will be the opportunity for explicit guidance by the programmer. If no guidance is given, the laboratory will doggedly pursue possible hypotheses but the search may be cut dramatically by human intervention.

Even with this assistance, the proposed transformations must be taken as tentative hypotheses to be explored. Few transformations always result in improvement. Many optimize one resource at the expense of others, while some transformations are of use only for certain regions of the data space. Hence, in general, it is necessary to continually verify that the transformed versions of the program are indeed improvements. Again, program analysis coupled with performance measurement tools will be employed.

In summary, the principal functions of the laboratory are choosing areas of the program to be optimized, carrying out pattern matching to determine the applicability of various transformations, performing these transformations, and arranging for explicit guidance by the programmer in this process. The laboratory consists of a set of components for carrying out these activities in concert.

COMPONENTS OF THE INITIAL LABORATORY

As we noted previously the laboratory is, essentially, one particular extension of ECL and that the internal representation for programs and data employed in ECL will be utilized for programs being manipulated (synthesized, transformed, proved to have some property, and so on). Here we will describe the several components in the initial laboratory—the several EL1 programs and/or ECL extensions which together constitute the initial laboratory. Before launching into the discussion, however, we want to note the influence of the work of Hewitt and Winograd on ours; the basic components of the laboratory have very much the flavor of the linguistic-independent components of Winograd's version of Hewitt's PLANNER system. Our discussion of the components is quite brief as our intention in this paper is to provide an overview of the laboratory and the general approach being taken. A detailed discussion of the current versions of the components is provided in a paper by Spitzen (see [Spitzen 71]).

Control

There is a top-level control path* which provides the interface between the user and the laboratory. It provides for input of the program to be manipulated and the associated performance criteria, if any. It then arranges that the program to be manipulated is put into a canonical form and sets up a parallel path which provides the manipulation strategy by calling for the appropriate theorem or theorems to be applied. The control path then provides for dialogue between the system and the user so that the system can request information from the user (e.g., verification that certain constraints hold in general, test data, and the like).

Programmable theorem prover

A "theorem" in the sense this term is used in the system consists of a condition which governs the applicability of the theorem, an antecedent, a consequent, and a recommendation list. Given some program structure, an instance of substructure matching the antecedent can be replaced by an equivalent substructure corresponding to the consequence so long as the condition holds. The recommendation list is basically a list of predicates which govern the use of other theorems to be used in manipulating the structure to match the antecedent in applying this one and it is the careful use of this "governor" which permits the theorem proving to operate in a practicable time frame. Note that in a very strong sense theorems are really programs effecting transformations which, through the conditions and recommendation lists (arbitrary EL1 predicates), can establish elaborate control and communication arrangements.

Data base

The theorems which constitute the basis for the transformations and manipulations performed by the system are housed in a data base. There is a continually growing static component of the data base which includes the "standard" theorems. In addition there may be a collection of theorems appropriate to certain particular investigations with which we will augment the data base at appropriate times. There is also a dynamic component which varies as a program is being manipulated. For example, if we commence dealing with the arm "$p_2 \Rightarrow e_2$"

* ECL provides multiprogramming; "path" is the ECL terminology for a parallel path of control.

of the conditional expression

$$[p_1 \Rightarrow e_1;\ p_2 \Rightarrow e_2;\ \ldots;\ p_n \Rightarrow e_n]$$

Then we would add the theorem appropriate to "$\neg p_1$" to the data base and arrange that it be removed when we get to the stage of treating the conditional expression as a whole in "higher level" manipulation of the program structure containing it.

The data base is quite large and, as various new facilities are added to the system, the data base will grow. Its size plus the fact that one wants to be able to select appropriate theorems (i.e., in accordance with a given recommendation list) from the data base efficiently, make it imperative that a certain amount of structure be imposed on the data base. Initially this structure basically amounts to a collection of "threads" through the data base, implemented by a combination of list structure, hash, and descriptor coding techniques. It is anticipated that getting an efficient structuring of the data base as it grows will pose a non-trivial research problem which we anticipate being attacked with the tools provided by the laboratory itself.

Pattern Matcher

The process of applying some theorem to some program structure involves matching the parts of the antecedent to acceptable corresponding parts of the structure; in general, of course, this will involve calls on other theorems to manipulate the structure into an appropriate format and/or the verification that certain conditions maintain. It is the pattern matcher which administers this activity; it will make tentative part-part matches, reject matches or attempt them in new ways when subsequent failure to match occurs, and so on.

Backtracking mechanism

The pattern matcher operates as a non-deterministic program in the sense that it makes provisional matches which must be "unmatched" and rematched when failure occurs to match some subsequent part of the antecedent to the program structure being manipulated. A backtracking mechanism must therefore be provided so that the effects of computations and variable bindings can be "undone." The method we use to accomplish this is to alter the basic EL1 evaluator so that, when backtrackable code segments are executed, any change to the environment is preceded by recording the appropriate modification to the environment which will undo the change in the event backtracking is required.

Measurement techniques

In addition to the usual facilities for inserting probes to provide measures of utilization of various functions and data elements provided in ECL and the usual ability to obtain measurements by sampling driven by a real-time clock there are two measurement components in the system which are less usual. Precise delineation of potential inefficiencies sometimes requires very exact timing data. Unfortunately it is usually impossible to get very fine-grained timing from most contemporary machines. Hence, the laboratory includes a machine language simulator for its host machine, i.e., a program which executes machine code interpretively and gathers statistics as requested. This permits programs to be run unchanged while collecting very precise data on the distribution of program execution time and storage requirements. This data, combined with that obtained by inserting measurement probes into the program permits performance measurements to be made to any level of detail. The second measurement component is an implementation of the "probability distribution computer" described by Nemeth.

GOALS OF THE LABORATORY

It must be stressed that the laboratory is intended for practical use, for the attainment of useful results which would be difficult to obtain without its assistance. It is the deliberate intention that one be able to approach it with a non-trivial program and obtain with it a significantly improved version. Such programs will include applications programs, system programs such as the EL1 compiler, as well as modules of the laboratory itself.

It appears that this will be achievable even with the simple approaches to be taken in the initial version of the laboratory. The use of *programmable* theorem provers and search techniques have made it possible to quickly endow the laboratory with considerable expertise, if not deep insight. On large production programs (which commonly are so complex that no one really understands their exact behavior) the laboratory may be expected to make significant improvements by more or less mechanical means. Such mechanical skill is precisely what is required to complement the high level programmer. Initially, the laboratory will serve as a programmer's assistant, suggesting possible areas of improvement and carrying out the transformations he chooses, but leaving the creative work to him. Even at this level the laboratory may serve better than the average junior programmer. Further, the initial labora-

tory will serve as a basis for developing more sophisticated control over the choice of transformations to be applied.

A topic which falls out as another application of this work is the modification or adaptation of programs. Given a program which does a certain job well, it is very common that it must be modified to handle a related task or work under changed conditions. Usually such modifications are done by hand, seldom with entirely satisfactory results. Hence, automation of the process is attractive. One method which has been occasionally proposed for performing this is to construct a system which when fed the existing program, deduces how it works, and then performs the specified modifications. This requires the system to dig the meaning and purpose out of the existing program—an operation which is difficult at best and perhaps impractically expensive. A solution which shows greater promise is to use the laboratory to combine man and machine. If programs are developed using the laboratory, then for any production program, there is an initial program from which it was derived. Given a statement of the new task to be accomplished, the programmer can make his modifications to the relatively transparent and simple initial program. This is still hand labor, but at a much higher level. Applying the laboratory to the optimization of the modified program results in a production program with the desired properties. This puts creative work in adapting a program into the hands of the programmer while freeing him from the drudge work.

It is anticipated that as the programmer gains experience with the system, he will develop his own set of optimization techniques. By making it convenient for him to add to the data base these transformations along with patterns for recognizing potential situations for their applications, we allow a certain degree of growth in the expertise of the laboratory. Given a clever population of users, the laboratory should grow to some considerable level of sophistication, at least in areas of interest to that user population. This scenario has, of course, its limitations. Merely expanding the data base would, in time, cause the system to flounder in a sea of possible but irrelevant transformations. Hence growth of the system must ultimately depend on significant improvements in global strategies, local heuristics, and theorem provers. In particular, the need for specialized theorem provers will very likely arise.

At present, it is not clear how to proceed in these directions, nor is this surprising. One of the purposes of the laboratory is to gain expertise in program manipulation, determine the limitation of current techniques, and advance to the point where the real problems can be seen clearly. The initial laboratory is a first step, but a significant step, in this direction.

BIBLIOGRAPHY

E A ASHCROFT (1970)
Mathematical logic applied to the semantics of computer programs
PhD Thesis Imperial College London

E A ASHCROFT Z MANNA (1970)
Formalization of properties of parallel programs
Stanford Artificial Intelligence Project Memo AI-110 Stanford University

R M BURSTALL (1970)
Formal description of program structure and semantics in first-order logic
Machine Intelligence 5 (Eds Meltzer and Michie)
Edinburgh University Press, 79-98

R M BURSTALL (1969)
Proving properties of programs by structural induction
Comp J 12 1 41-48

D C COOPER (1966)
The equivalence of certain computations
Computer Journal Vol 9 pp 45-52

R W FLOYD (1967)
Assigning meanings to programs
Proceedings of Symposia in Applied Mathematics American Mathematical Society Vol 19 19-32

R W FLOYD (1967)
Non-deterministic algorithms
JACM Vol 14 No 4 (Oct.)

B A GALLER A J PERLIS (1970)
A view of programming languages
Chapter 4 Addison-Wesley

C GREEN (1969b)
The application of theorem proving to question-answering systems
Ph D Thesis Stanford University Stanford California

C GREEN B RAPHAEL (1968)
The use of theorem-proving techniques in question-answering systems
Proc 23rd Nat Conf ACM Thompson Book Company Washington DC

P J HAYES (1969)
A machine-oriented formulation of the extended functional calculus
Stanford Artificial Intelligence Project Memo 62 Also appeared as Metamathematics Unit Report University of Edinburgh Scotland

C HEWITT (1971)
Description and theoretical analysis of plannar
Ph D Thesis MIT January

C B JONES P LUCAS
Proving correctness of implementation techniques
Symposium on Semantics of Algorithmic Languages Lecture Notes in Mathematics 188 New York

D M KAPLAN (1970)
Proving things about programs
Proc Princeton Conference on Information Science and System

D M KAPLAN (1967)
Correctness of a compiler for algol-like programs
Stanford Artificial Intelligence Memo No 48 Department of Computer Science Stanford University

J KING (1969)
A program verifier
Ph D Thesis Carnegie-Mellon University Pittsburgh Pa

J KING R W FLOYD (1970)

Interpretation oriented theorem prover over integers
Second Annual ACM Symposium on Theory of Computing Northampton Mass (May) pp 169-179
D C LUCKHAM N J NILSSON (1970)
Extracting information from resolution proof trees
Stanford Research Institute Artificial Intelligence Group Technical Note 32
Z MANNA (1969)
The correctness of programs
J of Computer and Systems Sciences Vol 3 No 2 119-127
Z MANNA (1969)
The correctness of non-deterministic programs
Stanford Artificial Intelligence Project Memo AI-95 Stanford University
Z MANNA J McCARTHY (1970)
Properties of programs and partial function logic
Machine Intelligence 5 (Eds Meltzer and Michie)
J McCARTHY (1963)
A basis for a mathematical theory of computation
Computer programming and formal systems (Eds Braffort and Hirschberg) Amsterdam North Holland pp 33-70
J McCARTHY J A PAINTER (1967)
Correctness of a compiler for arithmetic expressions
Mathematical Aspects of Computing Science Amer Math Soc Providence Rhode Island pp 33-41
R MILNER
An algebraic definition of simulation between programs
Stanford Artificial Intelligence Memo AI-142
A NEMETH
Unpublished; to be included in his Ph D Thesis Harvard University
J A PAINTER (1967)
Semantic correctness of a compiler for an algol-like language
Stanford Artificial Intelligence Memo No 44 (March) Department of Computer Science Stanford University
M S PATERSON (1967)
Equivalence problems in a model of computation
PhD Thesis Cambridge University
G ROBINSON L WOS
Paramodulation and theorem-proving in first-order theories with equality
Machine Intelligence Vol IV Ed by D Michie
J ROBINSON (1966)
A review of automatic theorem proving
Annual Symposia in Applied Mathematics Vol XIX
J T SCHWARTZ (1970-71)
Abstract algorithms and a set-theoretic language for their expression
Preliminary Draft First Part Courant Institute of Mathematical Sciences NYU
J M SPITZEN (1971)
A tool for experiments in program optimization
(in preparation)
R M SHAPIRO H SAINT (1969)
The representation of algorithms
Rome Air Development Center Technical Report 69-313 Volume II September
H A SIMON (1963)
Experiments with a heuristic compiler
JACM (October 1963) 482-506
J SLAGLE (1967)
Automatic theorem proving with renameable and semantic resolution
JACM Vol 14 No 4 (October 1967) 687-697
J SLAGLE (1965)
Experiments with a deductive question-answering program
Comm ACM Vol 8 No 12 (December) 792-798
H R STRONG JR (1970)
Translating recursion equations into flow charts
Proceedings Second ACM Symposium on Theory of Computing (May) pp 184-197
G J SUSSMAN T WINOGRAD (1970)
Micro-plannar reference manual
MIT AI Memo No 203 (July)
R J WALDINGER (1969)
Constructing programs automatically using theorem proving
PhD Thesis Carnegie-Mellon University (May 1969)
B WEGBREIT (1971)
The ECL programming system
Proc FJCC Vol 39
R WINOGRAD (1971)
Procedures as representative for data in a computer program for understanding natural language
Project MAC MIT MAC-TR-:4 February