Programming Languages

Thomas A. Standish Editor

Mechanical Program Analysis

Ben Wegbreit
Xerox Palo Alto Research Center

One means of analyzing program performance is by deriving closed-form expressions for their execution behavior. This paper discusses the mechanization of such analysis, and describes a system, Metric, which is able to analyze simple Lisp programs and produce, for example, closed-form expressions for their running time expressed in terms of size of input. This paper presents the reasons for mechanizing program analysis, describes the operation of Metric, explains its implementation, and discusses its limitations.

Key Words and Phrases: analysis of programs, performance analysis, execution time, execution behavior, difference equations, generating functions, list processing, Lisp, algebraic manipulation, programming languages, analysis of algorithms

CR Categories: 3.69, 4.22, 5.24, 5.25

Author's address: Xerox Palo Alto Research Center, 3180 Porter Drive, Palo Alto, CA 94304.

1. Introduction

One means of analyzing program performance is by deriving closed-form expressions for their execution behavior. This is an important facet of programming. In this paper, we discuss the mechanization of such analysis. We first outline the main problems to be addressed. We then describe a prototype system, Metric, which is able to analyze simple Lisp programs and produce closed-form expressions for their execution behavior in terms of properties of the input, e.g. execution time as a function of the length of the arguments. The Metric system has been implemented in Interlisp [19]; this implementation is discussed, with emphasis on system organization and general techniques. We conclude with a brief discussion of issues raised by this study.

Two limitations should be noted at the outset. One, standard halting-problem arguments show that no such system can be complete: execution time is not a decidable property of current programming languages. Two, the analysis of many algorithms requires considerable mathematical expertise; an expert system would necessarily include all the techniques in the monumental work of Knuth [9]. The former is an absolute limitation; the latter establishes a boundary beyond which interactive assistance from a programmer or analyst is required [5]. We are concerned in this paper with establishing mechanical program analysis as a desirable and feasible activity within these limitations.

Mechanical program analysis has three main applications:

1. As a software engineering tool: serving as an aid to the programmer in understanding how a program behaves.
2. In an automatic program synthesizer [6, 10, 12]. In general, there are many ways in which an axiomatic program specification can be realized; some knowledge of performance is required if a program synthesizer is to make a good choice.
3. In the compiling system for a very high-level language [15]. To the extent that a very high-level programming language statement avoids commitment of procedural steps, the issues of synthesis arise: there are many possible procedural renderings of a program and performance is the criterion to choose among logically equivalent alternatives (cf. [3] for amplification of this point).

There are a variety of measurement techniques [8, 7, 14] for obtaining execution profiles, i.e. plots of time spent in each program region when the program is run on sample data. While such profiles serve the needs of the first application reasonably well, they must be supplemented by analysis for the purposes of the other two applications. In particular, to optimize a program written in a very high-level language, the system must not only find where the program is spending resources but also determine why, i.e. analyze what it is doing there. Some closed-form representations of program behavior in these regions seems required if one is to go much beyond the classical [4] compiler optimization techniques. With a suitable closed-form representation, one has a start at finding the cause, in program terms, of poor functional behavior or unacceptably large coefficients.

The behavior of a program can be characterized by a set of properties: execution time on a particular machine, amount of storage used, size of its output, probability of its result satisfying a certain predicate, etc. Some of these properties (e.g. time) are of intrinsic interest; others (e.g. probabilities) are of interest principally because they are required in computing properties of intrinsic interest. We use the term *measure* generically to denote any of these properties.

Given a program and a specified measure, the problem of analysis is first to determine what properties of the data are most relevant to program performance under that measure and then to find a closed-form expression in terms of these properties. In general, an exact expression in terms of known properties of the input cannot be obtained, e.g. internal tests may depend on computed quantities having no simple relation to the input. Such tests are treated probabilistically, e.g. as Markov processes [1, 16] when the probability is constant and independent of prior history. Following Knuth [9], performance of a program under some measure can be expressed as a four-tuple $\langle min, max, mean, variance \rangle$. The term *performance* will be subsequently used strictly to denote such a four-tuple. We use the term *range* to denote the pair $\langle min, max \rangle$ and *moment* to denote the $\langle mean, variance \rangle$. A scalar S is an abbreviation for the performance $\langle S,S,S,0 \rangle$.

Metric is a prototype system, constructed to study the mechanization of program analysis; as such, it concentrates on certain key issues ignoring many peripheral ones. Its source language is essentially Lisp 1.0 as described in [11]. In the interest of brevity, we refer the reader to [11] and [20] for an explanation of list processing and Lisp. In presenting example programs, we use the following notation:

The empty list is denoted by $\{\}$.
A nonempty list is denoted by $\{f.r\}$ where f is the first element of the list and r is the list consisting of all elements except the first.
$CONS(f,r) = \{f\,.\,r\}$
$CAR(\{f\,.\,r\}) = f$
$CDR(\{f\,.\,r\}) = r$
$ATOM(x)$ is a predicate which is **true** if and only if x is not a list or is the empty list.
$NULL(x)$ is defined as $x = \{\}$.
Conditional expressions are written as:
if p_1 **then** e_1 **else if** p_2 **then** $e_2 \dots$ **else** e_n

Given the definition of a set of procedures, Metric attempts to produce analysis for their running times, number of *CONS* executed, number of list cells in their result, etc., as directed. In its current state, it can handle only simple programs such as those which might be used as introductory exercises in Lisp programming (e.g. append, reverse, nth, substitute, flatten, member, and union). However, the system is built on methods with general competence, and within its province, it has some degree of expertise. Subject to certain limitations, these methods are extendable to more complex programs and a richer set of data and control structures. In the conclusion, we outline how the extensions may be affected and discuss the limiting constraints.

This paper is divided into seven sections. Section 2 gives several examples of programs and their analysis by Metric. Section 3 is an overview of the system organization. Sections 4, 5, and 6 describe the three principal phases of the system: assigning local costs, analyzing recursion, and solving difference equations. Section 7 discusses the extensions of these techniques.

2. Examples

We begin with a set of examples which illustrate the sort of analysis that Metric can carry out. Here we are concerned only with what Metric can handle; subsequent sections discuss how.

A conceptually simple procedure for reversing the top level of a list is given by:

```
REVERSE(L) ≡
if NULL(L) then {}
else APPEND(REVERSE(CDR(L)),
            CONS(CAR(L), {}))

APPEND(X,Y) ≡
if NULL(X) then Y
else CONS(CAR(X),
          APPEND(CDR(X),Y))
```

Metric determines that the execution time for *REVERSE*(e) is given by $c_0 + c_1 \cdot n + c_2 \cdot n^2$ where n is the *length* of e (i.e. the number of list cells in the *cdr* direction) and the c_i's are implementation constants. In its normal mode of operation, Metric computes such implementation constants symbolically, as linear arithmetic expressions of the form $n_i \cdot e_i + \cdots + n_k \cdot e_k$ where the n_i's are real numbers and the e_i's denote executions of elementary procedures. For example[1]:

$$c_2 = (1/2) \cdot null + (1/2) \cdot cdr + (1/2) \cdot cons + (1/2) \cdot car + (1/2) \cdot fncall + 2 \cdot vref.$$

The lowercase spelling of a primitive operator stands for that operation; *fncall* denotes the action required to invoke a nonprimitive procedure; *vref* denotes access to a variable; and *cref* denotes access to a constant. Generally, it is convenient to ignore the distinction between the costs of *CAR* and *CDR*; *cr* is then used to denote

[1] The appearance of nonintegral coefficients such as 1/2 may seem puzzling. The reason is as follows: the execution time is most naturally expressed in the form $d_0 + d_1 \cdot n + (1/2) \cdot d_2 \cdot n \cdot (n - 1)$, where the d_i's are linear arithmetic expressions with integral coefficients (cf. Section 6). Going from this natural form to a polynomial creates the nonintegral coefficients in the c_i's above.

either. The other two constants are given by:

$$c_1 = (3/2)\cdot null + cr + (1/2)\cdot cons + (3/2)\cdot fncall + 3\cdot vref + cref$$
$$c_0 = null + vref + cref.$$

This symbolic representation of execution time was chosen as being the simplest machine-independent form. By assigning values to the elementary operation symbols, one can obtain number of *CONS* executed, number of memory references, or microseconds for the computation time under an interpreter or simple compiler.[2]

FLAT is a procedure which flattens a list, constructing a one-level list of the atoms in a possibly multilevel list, e.g. $FLAT(\{A.\{\{B.C\}.D\}\}) = \{A,B,C,D\}$. One way of doing this uses a doubly-recursive auxiliary procedure *FLAT*2:

```
$FLAT(L) \equiv FLAT2(L, \{\})$

$FLAT2(X,Y) \equiv$
if $ATOM(X)$ then $CONS(X,Y)$
else $FLAT2(CAR(X),$
            $FLAT2(CDR(X),Y))$
```

Metric determines that the time for $FLAT(L)$ depends on the *size* of L, i.e. the number of list cells.[3] Specifically, the time is found to be $c_0 + c_1\cdot s$, where s is the size of L, and:

$$c_1 = cons + 2\cdot fncall + 2\cdot cr + 2\cdot atom + 7\cdot vref$$
$$c_0 = cons + atom + fncall + 4\cdot vref + cref.$$

In analyzing a program under one measure, one or more other measures are typically applied in the decomposition. For example, in computing *time* of $REVERSE(FLAT(APPEND(P,Q)))$, it is found that *length* of the argument to *REVERSE* is needed. Analyzing *FLAT* under the length measure, Metric obtains $length(FLAT(L)) = 1 + size(L)$. Continuing, it finds that $size(APPEND(P,Q)) = size(P) + size(Q)$. Hence, the length of the argument to *REVERSE* is found to be $1 + size(P) + size(Q)$, giving one constituent of the time.

Length and *size* are structural properties of variables, analogous to dimensions of arrays or number of records in a file. Metric attempts to express program behavior in terms of these. When this is not possible, due to internal tests not related to structural properties, Metric expresses the analysis as a performance in which probabilities of unanalyzable tests appear as parameters. For example, the number of times that atom X appears in the top level of list L is computed by:

```
$COUNT(X,L) \equiv$
if $NULL(L)$ then 0
else if $X = CAR(L)$
  then $ADD1(COUNT(X,CDR(L)))$
else $COUNT(X,CDR(L))$
```

The probability of the test $X = CAR(L)$ succeeding is needed to obtain the time performance of *COUNT*. However, the operation "=" is primitive and cannot be further analyzed. The system can proceed no further without using additional knowledge supplied to it. If, on its input data base, it finds that the test "=" in *COUNT* may be treated as constant, it assigns a symbolic probability, say p, its value to be determined by measurement. Metric then is able to determine that the time performance is given by:

$$\langle c_0 + c_1 n, c_0 + c_2 n, c_0 + c_3 n, c_4 n\rangle$$

where $n = length(L)$ and

$$c_0 = null + cref + vref$$
$$c_1 = fncall + null + eq + 2\cdot cr + 5\cdot vref$$
$$c_2 = add1 + 2\cdot fncall + null + eq + 2\cdot cr + 5\cdot vref$$
$$c_3 = p\cdot add1 + p\cdot fncall + fncall + null + eq + 2\cdot cr + 5\cdot vref$$
$$c_4 = p\cdot add1^2 + 2\cdot p\cdot add1\cdot fncall + p\cdot fncall^2 - p^2\cdot add1^2 - 2\cdot p^2\cdot add1\cdot fncall - p^2\cdot fncall^2$$

An instructive counterpart is *UNION* which assumes its arguments are lists of nonrepeating atomic symbols and forms their set union:

```
$UNION(X,Y) \equiv$
if $NULL(X)$ then $Y$
else if $MEMBER(CAR(X),Y)$
  then $UNION(CDR(X),Y)$
else $CONS(CAR(X),$
            $UNION(CDR(X),Y))$
```

Here the test to be treated probabilistically is a defined procedure:

```
$MEMBER(Z,L) \equiv$
if $NULL(L)$ then false
else if $Z = CAR(L)$ then true
else $MEMBER(Z,CDR(L))$
```

so that $probability(MEMBER(CAR(X),Y))$ is a derived expression which can be expressed in terms of other quantities. Analyzing *MEMBER* using *probability* of the result being **true** as the measure, Metric obtains $1 - (1 - a)^m$ where $a = probability(Z = CAR(L))$ and $m = length(L)$. Using this, the length of $UNION(X,Y)$ is found to be:

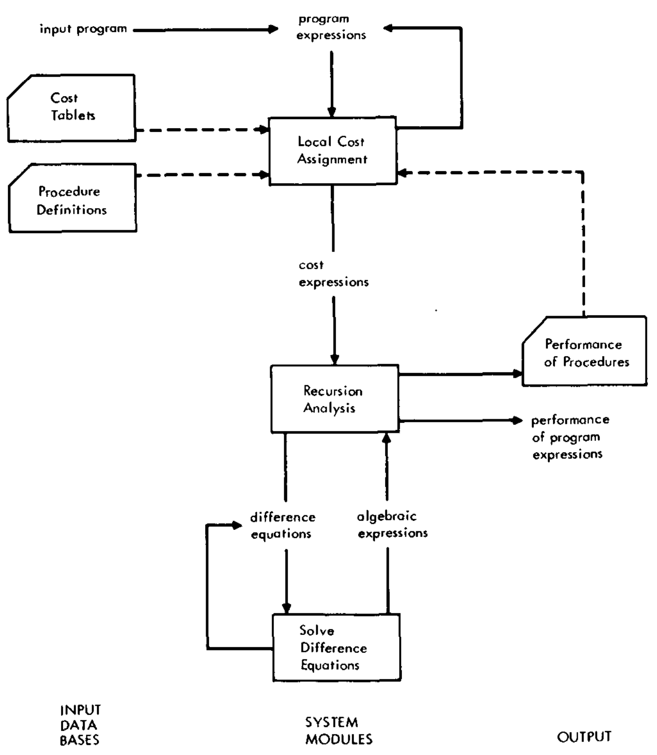
$$\langle m, m + n, m + n\cdot(1 - a)^m, n\cdot(1 - a)^m - n\cdot(1 - a)^{2\cdot m}\rangle$$

where $m = length(Y)$ and $n = length(X)$.

[2] This does not provide a satisfactory basis to assess computation time under an optimizing compiler, since no account is taken of possible transformations to the execution sequence (e.g. common subexpression elimination, and removal of invariant computation from recursive calls). To handle this properly, it would be necessary to apply the local cost assignment phase discussed in Section 4, not to the source program, but rather to the intermediary code or to the generated code itself. This would complicate the implementation both directly (program representation is usually more complex) and indirectly (compiler interfaces must be established) but presents few conceptual problems. We return to this in Section 7.

[3] Size may be defined by $size(L) \equiv$ **if** $NULL(L)$ **then** 0 **else** $1 + size(CAR(L)) + size(CDR(L))$, and *length* analogously.

Fig. 1. Structure of Metric.



3. System Structure

The overall organization of Metric is shown in Figure 1. Solid lines show the flow of control through the system; dashed lines show the use of data, either input data or previously computed results. The input data bases are a set of procedure definitions and a set of tables used to establish the symbolic cost of elementary operations. Direct input is a program expression and one of the known measures to use in its analysis. The output is the specified analysis plus a data base of the performance of procedures analyzed in this process. Subsequent calls on the system retrieve the results of prior analysis when applicable. Hence, the system can be supplied performance information but will work from definitions if that is unavailable.

Analysis of a program expression takes place in three phases:

Phase 1: Local cost assignment. A *cost* is assigned to each constituent as follows: Primitive operations (e.g. *CAR*) and language overhead activities (e.g. *function call*) are assigned costs as specified by the local cost tables. Defined operations are assigned the cost of their definition except that recursive procedure calls are detected and specially marked. The analysis of a nonrecursive procedure is determined by the composition of local costs; a recursive procedure is passed on to the next phase.

Phase 2: Recursion analysis. The procedure is symbolically evaluated to determine how the recursion variables change from one call to the next. This gives the recursive structure of the computation sequence. Next, the computation sequence is projected into the integers by constructing a set of difference equations which model the list structure manipulation carried out by the recursive calls.

Phase 3: Solution of difference equations. One or more of the following techniques are used to obtain closed-form expressions: direct summation, pattern matching, elimination of variables, best-case/worst-case analysis, and differentiation of generating functions.

The solution to the difference equations gives an expression for the performance of the originating recursive procedure. This is simplified, put into functional form, and stored under the pair ⟨*procedure*, *measure*⟩ for subsequent retrieval. A procedure thus analyzed has its cost given by the stored functional form. The next three sections explain these three phases in more detail.

4. Assigning Local Costs

The local cost assignment phase maps a program expression into a symbolic *cost expression* which specifies its cost under a given measure. Measures may be broadly grouped into two classes—*cumulative* and *noncumulative*—depending on how the arguments to a procedure appear in the measure of that procedure. Cumulative measures (e.g. *time*) treat nested procedure calls as additive; for example, the time to *APPEND* (*REVERSE*(*P*), *CONS*(*Q*,*R*)) is the sum of the times required to *REVERSE*(*P*), *CONS*(*Q*,*R*) and *APPEND* the results. Noncumulative measures (e.g. *length*) ignore inner procedure calls except when a property of their result is explicitly needed; for example, the *length* of *CONS*(*REVERSE*(*X*), *APPEND*(*Y*,*Z*)) does not depend upon the first argument to *CONS*, so *REVERSE* can be totally ignored. Cumulative measures describe resource expenditure: time, number of *CONS* (and I/O activity if I/O was considered). Noncumulative measures describe the result of a procedure independent of how that result is obtained: size, length, probability of a certain specified result (and data type if data types were considered).

To maintain uniform notation for expressions involving the two classes of measures, ⟨*measure*⟩ (⟨*procedure application*⟩) is always interpreted as the cost of ⟨*procedure application*⟩ under ⟨*measure*⟩ *after* all arguments have been evaluated. Thus, the total time to compute the entire program expression *APPEND* (*REVERSE*(*P*), *CONS*(*Q*,*R*)) is expressed as: *time* (*REVERSE*(*P*)) + *time*(*CONS*(*Q*,*R*)) + *time*(*APPEND*(*REVERSE*(*P*), *CONS*(*Q*,*R*))) + *time to access* *P*, *Q*, and *R*.

Procedures may be grouped into three classes: (1) *primitive*—built-in operations of the language (e.g. *CDR*); (2) *fixed*—defined procedures containing no recursive calls or invocations of recursive procedures (e.g. *NOT*); and (3) *closed*—procedures which call themselves (directly recursive) or invoke other closed procedures.

4.1 Primitive Procedures

Primitive procedures and constants are assigned costs as specified by the local cost tables. Depending on the measure and procedure, these may be reals, scalar expressions, or performance four-tuples. For example, the constant { } is assigned cost 0 under the measure length. Similarly, under the size measure, $size(CONS(e_1,e_2)) = 1 + size(e_1) + size(e_2)$. Under the measure *time*, the usual assignment is a symbolic expression, e.g. $time(CONS(e_1, e_2)) = cons$. However, this may be reduced to a more elementary form, such as memory references or microseconds on a given machine. In some cases, performance four-tuples may be used. For example, in an implementation with *cons*-paging [2] the number of instructions executed to perform a *CONS* is variable; similarly a read operation would be modeled by its performance $\langle min, max, mean, variance\rangle$. The use of performance four-tuples in this way allows the system to obtain statistical analysis of programs whose primitive (i.e. unanalyzable) operations have variable costs.

4.2 Fixed Procedures

Fixed procedures are assigned costs by analyzing their bodies. The body of a fixed procedure may be decomposed into the disjunction of one or more execution paths $r_1, \ldots, r_n$ where internal tests choose the appropriate path. Since each execution path consists only of primitive and other fixed procedures, local cost assignment is used directly to assign a cost c_i to each path r_i. Similarly, local analysis under the measure probability is used to obtain the probability p_i of taking that path.[4] The performance is then:

$$\langle \underset{i}{minimum}(min(c_i)),\ \underset{i}{maximum}(max(c_i)),\ \sum_i(p_i\cdot mean(c_i)),\ \sum_i(p_i\cdot variance(c_i) + p_i\cdot mean(c_i)^2) - (\sum_i(p_i\cdot mean(c_i)))^2\rangle.$$

This is complicated by the need to express the results symbolically, e.g. *cons* and $(3\cdot cr + atom)$ are incomparable. Their minimum is therefore expressed as: $minimum\,(cons, 3\cdot cr + atom)$. When the performance of a fixed procedure has the form $\langle S,S,S,0\rangle$ (because there is only one path or all paths have equal cost), it is frequently useful to suppress detail and express the cost of that procedure as a single entity. For example, time of *NOT* defined as $NOT(X) \equiv$ **if** X **then false else true** is represented by the simple symbolic expression *not*.

As an example, consider *EQ*3 which tests whether its three arguments are identical:

```
EQ3(X,Y,Z) ≡
if NOT(X = Y) then false
else if NOT(X = Z) then false
else true
```

*EQ*3 has three branches with probabilities $(1 - a)$, $(1 - a)\cdot a$, and a^2 respectively, where a is the probability of *EQ* returning *T*. Under the measure time, the costs are

$$c_1 = 2\cdot vref + eq + fncall + not + cref,$$

$$c_2 = 4\cdot vref + 2\cdot eq + 2\cdot fncall + 2\cdot not + cref,$$

and c_2 respectively; hence, performance is:

$$\langle c_1,\ c_2,\ c_1 + a\cdot(c_2 - c_1),\ c_1^2 + a\cdot c_2^2 - a\cdot c_1^2 - (c_1 + a\cdot(c_2 - c_1))^2\rangle.$$

Since terminal conditions of these paths are **false, false**, and **true** respectively, the probability of *EQ*3 returning the result **true** is determined to be a^2. This would be computed, by analyzing *EQ*3 under the measure *probability*, if *EQ*3 appeared as the test in another procedure.

4.3 Closed Procedures

Closed procedures are assigned costs by processing their definitions, assigning local costs to primitive and fixed constituents as described above, but giving special treatment to directly recursive calls and calls on other closed procedures. We consider direct recursion first. *The essential idea is to map a recursive procedure P into a new recursive procedure whose value is the cost of P.* Recursive calls are marked with the measure being used in the current analysis. That is, if the definition of P being analyzed under measure M contains a subexpression $P(e_1, \ldots, e_n)$, then, when the cost assignment phase encounters this and detects recursion, it returns the symbolic expression $M(P(e_1, \ldots, e_n))$. For example, using the definition of *APPEND* given in Section 2:

```
length(APPEND(X,Y)) ≡
if NULL(X) then length(Y)
else 1 + length(APPEND(CDR(X),Y))
```

Recursion analysis, discussed in the next section, uses this to determine, symbolically, how the arguments are modified from call to embedded call.

Calls from a closed procedure to another closed procedure are handled by analyzing the called procedure recursively to obtain its cost—possibly under a different measure. For example, consider determining the time of *REVERSE*. An intermediary representation

[4] In computing this probability, the system treats all tests as statistically independent, so the probability of a sequence of choices is computed to be the product of their independent probabilities. This simplifying assumption is often invalid. Merely detecting the possibility of nonindependent tests would not be difficult: it suffices to be conservative and report possible dependency whenever the analyzer cannot guarantee independence. However, going further and analyzing the dependencies is a fundamental, deep problem and beyond the scope of the present paper. We return to this issue in the conclusion.

would be:

if $NULL(L)$ **then** $null + vref + cref$
else $null + 3 \cdot vref + cref + 2 \cdot cr + cons + 2 \cdot fncall$
$time(REVERSE(CDR(L))) +$
$time(APPEND(REVERSE(CDR(L)),$
$CONS(CAR(L), \{\})))$

Since the symbolic expression $time(APPEND(REVERSE(CDR(L)), CONS(CAR(L), \{\})))$ invokes another closed procedure, the local analysis phase invokes the system recursively to obtain a closed-form expression for this in terms of primitive operations. This is carried out in the following steps.
(i) *APPEND* is analyzed under the measure *time*. The system is thereby invoked recursively; it runs through all three phases in carrying out this analysis and produces the answer: $time(APPEND(X,Y)) = c_0 + c_1 \cdot length(X)$ and computes the constants c_0 and c_1. The process by which the answer is obtained is developed in this and the next two sections. Here, it suffices to continue with the result.
(ii) Since the *length* of the first argument to *APPEND* is needed, *REVERSE* is analyzed under the measure *length*.

$length(REVERSE(L)) \equiv$
if $NULL(L)$ **then** 0
else $length(APPEND(REVERSE(CDR(L)),$
$CONS(CAR(L), \{\})))$

Again, instead of returning the result in this form, Metric attempts to determine the value of $length(APPEND(REVERSE(CDR(L)), CONS(CAR(L), \{\})))$.
(iii) To do so, it analyzes *APPEND* under the measure *length*. The phase (1) result is shown above. Phases (2) and (3) eventually result in: $length(APPEND(X,Y)) = length(X) + length(Y)$.
(iv) To use the result of step (iii) in step (ii), $length(CONS(CAR(L), \{\}))$ and $length(REVERSE(CDR(L)))$ are needed. The former is 1. The latter is a use of *length* (*REVERSE*) while analyzing *REVERSE* under the measure *length*; consequently, it is represented as $length(REVERSE(CDR(L)))$.
(v) Substituting (iii) and (iv) into (ii), length of *REVERSE* is expressed as

if $NULL(L)$ **then** 0
else $1 + length(REVERSE(CDR(L)))$

From this, subsequent phases find that $length(REVERSE(L)) = length(L)$.
(vi) The result of (v) is combined with the result of (i), to obtain: $time(APPEND(REVERSE(CDR(L)), CONS(CAR(L), \{\}))) = c_0 + c_1 \cdot (length(L) - 1)$ where the c_i's are the constants from step (i). This, then, is used to obtain an expression for the time of *REVERSE*.

This process wherein the analysis of a procedure under one measure invokes the analysis of called procedures under different measures is somewhat analogous to the generation and proof of subsidiary lemmas in automatic program verification. In Metric, it is used frequently. In addition to cases like the above, a defined predicate encountered as the *conditional*-test in a procedure definition is analyzed for probability of its returning the value **true**. It is useful in this regard to treat each measure as imposing an interpretation (i.e. model) on the primitive operator names and local cost assignment as evaluation in this model. Local cost assignment maps fixed procedures into fixed cost expressions and recursive procedures into recursive cost expressions.

When Metric discovers that one recursive procedure calls another, it temporarily suspends analysis of the first, analyzes the second to obtain its cost in closed form, and substitutes a closed-form expression in place of the call to the second procedure. Called procedures are thus systematically eliminated from subsequent consideration. Hence, the recursive cost expressions produced by the local cost phase contain only one function letter, which simplifies the construction of difference equations in the next phase. This elimination method works only for certain call disciplines: define a set of procedures to be *well nested* if, whenever *A* calls *B*, no procedure called by *B* calls *A*. Note the analogy with well-nested loops. Also, note the specific relation that any well-nested iterative loop structure can be turned into a set of well-nested recursive procedures—each label is turned into a procedure name, and each backward **goto** into a procedure call. The elimination method works only on well-nested procedures. If the nesting structure is viewed as a tree, the processing order corresponds to a prefix walk.

5. Analyzing Recursion

The recursion analysis phase attempts to map the recursive cost expression for a procedure into a set of difference equations whose solution gives the performance of the original procedure. This takes place in three steps: (1) reduction to normal form; (2) construction of recursion equations by symbolic evaluation and case discrimination; and (3) projection into the integers.

5.1 Reduction to Normal Form

A recursive cost expression for the procedure *P* under measure *M* contains one or more execution paths which include $M(P(arg_1, \ldots, arg_n))$ and one or more paths free of recursion. The *normal form* for such a cost expression is a conditional:

if p_1 **then** c_1 **else if** p_2 **then** c_2 **else if** . . . **else** c_k

where the c_i's are free of conditionals.

The recursive cost expression for any pure Lisp procedure can be reduced to this form by moving all tests backward along the execution path and replacing embedded conditionals by conjunctions of the outer tests and inner tests. In the case of well-nested procedures analyzed by the elimination method of the previous section, the only nonprimitive procedure in the c_i's is P, the procedure being analyzed.

To simplify subsequent processing, it is desirable to eliminate all argument positions which are constant in all uses or which are manifestly irrelevant to the value of the cost expression. When an argument position is thus eliminated, the corresponding formal parameter is treated as a free variable. For example, to illustrate the elimination of constant argument positions, consider *SUBST* which substitutes X for the atom Y in Z

```
SUBST(X,Y,Z) ≡
if ATOM(Z) then if Y = Z then X
                else Z
else CONS(SUBST(X,Y,CAR(Z)),
          SUBST(X,Y,CDR(Z)))
```

Each recursive call on *SUBST* uses X as the first argument while X is also the first parameter in the defining form. Hence, the first argument position is constant over the course of recursion and may therefore be eliminated. The formal parameter X is then treated as a free variable. Similar considerations apply to the formal parameter Y and the second argument position. Thus, under the measure size, the cost has normal form:

```
size(SUBST(Z)) ≡
if ATOM(Z) & Y = Z then size(X)
else if ATOM(Z) then size(Z)
else 1 + size(SUBST(CAR(Z)))
       + size(SUBST(CDR(Z)))
```

An argument position may vary during the recursion and still be irrelevant to the value of the cost expression. Define a *relevant* argument position as follows: A formal parameter which appears in a nonrecursive cost expression is relevant, since the value of that cost expression depends on it; hence, the corresponding argument position is relevant. A formal parameter which appears in a test is relevant if the test actually depends on its value; hence, the corresponding argument position is relevant. Finally, a formal parameter is relevant if it appears as an argument in a recursive call within a position previously labeled as relevant; again, the argument position corresponding to the formal parameter is then relevant. An argument position is manifestly *irrelevant* if it is not shown to be relevant by the above rules. For example, consider the time of *FLAT2* defined in Section 2. Its normal form before parameter elimination is:

```
time(FLAT2(X,Y)) ≡
if ATOM(X) then $c_0$
else $c_1$ + time(FLAT2(CDR(X),Y))
    + time(FLAT2(CAR(X), FLAT2(CDR(X),Y)))
```

Although the second argument to *FLAT2* varies—being Y in the definition and $FLAT2(CDR(X),Y)$ in one recursive call—its value is manifestly irrelevant to the value of the cost function. Parameter elimination results in:

```
time(FLAT2(X)) ≡
if ATOM(X) then $c_0$
else $c_1$ + time(FLAT2(CDR(X)))
       + time(FLAT2(CAR(X)))
```

5.2 Construction of Recursion Equations

The second step constructs a set of recursion equations, converting the cost expression from procedural to declarative form. We begin with an example, continuing the above processing of *SUBST*. Let $E(Z) = size(SUBST(Z))$, let a_i be some unspecified atom, let s_1 and s_2 be unspecified values. Then $size(SUBST(Z))$ is defined by the recursion equations:

$$E(a_1) = size(X) \textbf{ when } Y = Z; \quad 0 \textbf{ when } Y \neq Z$$
$$E(\{s_1.s_2\}) = 1 + E(s_1) + E(s_2)$$

This is obtained by employing two processes simultaneously: *symbolic evaluation* and *case discrimination*. We consider these in turn.

Symbolic evaluation constructs a partial model of the data structures and values as specified by an execution path; it uses this model to partially evaluate subsequent expressions on that path. The model is represented in a symbolic association list, *alist*, which stores the values of variables and expressions as determined by tests: the left hand side of each conditional alternative is given an input *alist* and generates two output *alists*: with its truth (falsity) conjoined as a new binding for its Yes (No) output branch. The Yes output branch is used in evaluating the right hand side of the alternative; the No output branch is given as input to the next conditional alternative. In the above example, $ATOM(Z)$ adds $(Z = a_1)$ to the Yes *alist* and $(Z = \{s_1.s_2\})$ to the No *alist*. By convention, a_i's stand for unspecified atoms, s_i's for unspecified S-expressions, and n_i's for unspecified nonnegative integers.

Each form encountered during symbolic evaluation is evaluated so far as possible by using the information in the current *alist*. In outline, partial evaluation of a form using an *alist* proceeds as follows. A variable is replaced by its *alist* binding (e.g. Z by a_1 on the Yes arm after $ATOM(Z)$) or by itself if no binding is present. $CONS(e_1, e_2)$ is replaced by $\{e_1.e_2\}$; $CDR(\{e_1.e_2\})$ is replaced by e_2, $size(a_i)$ is replaced by 0, etc. Invocations of defined procedures are replaced by expressions for their definition (copy rule) up to the first recursive call, which evaluates to a (recursive) function application: dummy function symbol applied to the partial evaluation of the arguments. Symbolic evaluation only affects the right hand sides of conditional alternatives, i.e. their symbolic costs.

Case discrimination converts an initial sequence of conditional left hand sides $p_1, \ldots, p_k$ into a pattern P_k which describes the situation in which the kth conditional alternative will be chosen. The pattern p_k is divided into two parts: a structural portion, e.g. $E(a_1)$, and a **when** qualification, e.g. **when** $(Y = Z)$. The structural portion models that aspect of the arguments on which recursion is performed. The relevant properties in the case of list structure is distinguishing $\{\}$, atomic, and dotted pairs; in the case of dotted pairs, the system models as many levels as are manifest from the program tests. For example, on the path which takes the No branches of $NULL(X)$ and $NULL(CDR(X))$, the binding for X establishes that $X = \{s_1 . \{s_2 . s_3\}\}$ meaning: the *car* of X is some S-expression while its *cdr* is a dotted pair. In forming a set of recursion equations, the structural portion becomes the left hand side while the conditional alternative and **when** qualification become the right hand side. Right hand sides with identical left hand sides are grouped together.

The effect of symbolic evaluation combined with case discrimination is illustrated by the following recursion equations:

(1) Let $E(L) = time(REVERSE(L))$. Then

$E(\{\}) = d_0$

$E(\{s_1 . s_2\}) = d_1 + d_2 \cdot length(s_2) + E(s_2)$

for appropriate constant d_0, d_1, and d_2.

(2) Let $E(L) = length(UNION(L,Y))$, where Y is treated as a free variable. Then

$E(\{\}) = length(Y)$

$E(\{s_1 . s_2\}) = E(s_2)$ **when** $MEMBER(s_1,Y)$;

$1 + E(s_2)$ **when** $\sim MEMBER(s_1,Y)$.

(3) Let $E(X) = time(FLAT2(X,Y))$, where Y has been dropped, since step (i) finds that it is manifestly irrelevant. Then

$E(a_1) = c_0$

$E(\{s_1 . s_2\}) = c_1 + E(s_1) + E(s_2)$.

(4) Under the measure length, however, Y is quite relevant to $FLAT2$. Hence, let $E(X,Y) = length(FLAT2(X,Y))$. Then

$E(a_1, Y) = 1 + length(Y)$

$E(\{s_1 . s_2\}, Y) = E(s_1, E(s_2, Y))$.

5.3 Projection onto the Integers

The final step in analyzing recursion is mapping the recursion equations where the arguments are list structures into a set of difference equations where the arguments are integers. Define $E(arg_1, \ldots, arg_n)$ to be $F(b_1, \ldots, b_n)$ where each b_i is some integer valued function of arg_i; b_i is said to be an *abstraction* of arg_i. The abstractions are chosen such that: (a) the replacement of $E(arg_1, \ldots, arg_n)$ by $F(b_1, \ldots, b_n)$ can be done consistently, (b) all variables which are not integer-valued are replaced (except from **when** qualifications which are left unaltered). The current system uses only the abstraction's length and size.[5] To a first approximation, length is used if each recursive form involves only some nth *cdr* of the input, while size is used if some recursive form depends on *car* and *cdr* links. For example, the first two recursion equations above depend only on s_2, since s_1 is ignored. As this is the complete recursive description of E, the dependence only on s_2 carries to all levels, i.e. only the length of the argument to E is relevant. Hence, a new function F is defined, which makes this dependence explicit, $F(length(L)) = E(L)$. Since $length(\{s_1 . s_2\}) = 1 + length(s_2)$, the corresponding difference equations are:

(1′) Let $F(length(L)) = E(L)$.

$F(0) = d_0$

$F(n_2 + 1) = d_1 + d_2 \cdot n_2 + F(n_2)$.

(2′) Let $F(length(L)) = E(L)$.

$F(0) = length(Y)$

$F(n_2 + 1) = F(n_2)$ **when** $MEMBER(s_1,Y)$;

$1 + F(n_2)$ **when** $\sim MEMBER(s_1,Y)$.

Where both s_1 and s_2 appear as arguments to E recursively, the dependency is on size. A new function F is therefore defined as $F(size(X)) = E(X)$. Since $size \cdot (\{s_1 . s_2\}) = 1 + size(s_1) + size(s_2)$, the difference equation corresponding to (3) is:

(3′) Let $F(size(X)) = E(X)$.

$F(0) = c_0$

$F(n_1 + n_2 + 1) = c_1 + F(n_1) + F(n_2)$.

The appearance of an explicit length or size of an argument forces the abstraction of that argument position. Thus, in case (4):

(4′) $F(size(X), length(Y)) = E(X,Y)$.

$F(0, m) = 1 + m$

$F(n_1 + n_2 + 1, m) = F(n_1, F(n_2, m))$.

Note that the abstraction to the integers treats only the structural part of the pattern in the recursion equations. The **when** qualifications remain unchanged—to be used in subsequent processing.

6. Solving Difference Equations

The final phase solves difference equations such as the above to produce closed-form expressions. Difference equations may be considered in two groups depending on the absence or presence of **when** qualifications. Those without qualifications can have exact solutions. **When** qualifications give rise to performance expressions for which the *range* is obtained by considering best and worst cases and the *moments* are obtained from the derivatives of generating functions.

[5] Other possible abstractions include *car*-length, *max*-length (maximum path along any combination of *car* or *cdr* links), and *min*-length. Adding these to the system would not be difficult. More difficult but essentially understood is how to extend these to a language with multiple record types; for example, in such a language, each pointer field of a record defines a separate length class, etc. What is not well understood is how to synthesize an abstraction from the program when the correct one is not already known by the system; this is currently being studied.

6.1 Unqualified Difference Equations

Many of the unqualified difference equations can be solved very simply. For example, consider the difference equations for the time of *REVERSE*:

$$F(0) = d_0$$
$$F(n + 1) = d_1 + d_2 \cdot n + F(n).$$

This may be summed directly:

$$F(n) = d_0 + d_1 \cdot n + (1/2) \cdot d_2 \cdot n \cdot (n - 1).$$

Rewriting this as a polynomial in n,

$$F(n) = d_0 + (d_1 - d_2/2) \cdot n + (1/2) \cdot d_2 n^2.$$

Letting $c_0 = d_0$, $c_1 = (d_1 - d_2/2)$, and $c_2 = d_2/2$, the time expression given in Section 2 is obtained:

$$F(n) = c_0 + c_1 \cdot n + c_2 \cdot n^2.$$

Similarly, the system:

$$F(0) = c_0$$
$$F(n + 1) = c_1 + b \cdot F(n)$$

has the solution:

$$F(n) = c_1/(1 - b) + b^n(c_0 - c_1)/(1 - b)).$$

A related class of simple difference equations arises from programs where some variables are being built up (*CONS*, or *ADD*1) while other variables are being decomposed (*CDR*, or *SUB*1). For example, a procedure for reversing a list in linear time is given by:

```
REV(L) ≡ REV2(L, {})
REV2(X,Y) ≡
if NULL(X) then Y
else REV2(CDR(X),
         CONS(CAR(X),Y))
```

Computing length of *REV*2 gives rise to the difference equations:

$$F(0, m) = m$$
$$F(n + 1, m) = F(n, m + 1)$$

with solution $F(n,m) = n + m$.

The use of a size abstraction caused by simultaneous *car* and *cdr* recursion creates complications. Consider, for example, the difference equations for length of *FLAT*2:

$$F(0, m) = 1 + m$$
$$F(n_1 + n_2 + 1, m) = F(n_1, F(n_2, m)).$$

The appearance of two variables, such as $n_1 + n_2$ in an argument position implies potential indeterminacy since, in general, the value of the right hand expression depends on the particular choice of n_1 and n_2. Were this the case, it would be necessary to average over all choices of (i,j) pairs weighted by their computed or measured frequency. However, for a common class of programs, this is not the case—the value of the right hand side depends only on the sum $n_1 + n_2$, not on the particular values of n_1 and n_2. The system first guesses that this simple situation occurs. Under this hypothesis, it is free to consistently substitute for either n_1 or n_2 on the right and left hand sides. It chooses a constant which simplifies the problem—here, $n_1 = 0$—since this is a known base case. The result:

$$F(0, m) = 1 + m$$
$$F(n_2 + 1, m) = 1 + F(n_2, m)$$

is then readily solved: $F(n,m) = m + n + 1$. Finally, the guess is checked. Here,

$$F(1 + n_1 + n_2, m) = 2 + n_1 + n_2 + m,$$

so the guess is confirmed.

6.2 Qualified Difference Equations

If there is a **when** qualification, then a performance must be computed. We begin with an example. The difference equations for time of *MEMBER* are:

$$F(0) = c0$$
$$F(n + 1) = c1 \textbf{ when } X = CAR(Y);$$
$$c2 + F(n) \textbf{ when } X \neq CAR(Y).$$

$\langle min, max \rangle$ is obtained by best-case/worst-case analysis. $Min(F(n)) = minimum(c1, c0 + n \cdot c2)$, while $max(F(N)) = c2 \cdot (n - 1) + maximum\ (c1, c0 + c2)$. To obtain the moments $\langle mean, variance \rangle$, Metric uses generating functions.[6] Let $a = probability(X = CAR(Y))$. Let p_k be the probability that $F(n)$ has value k. Letting z be the formal variable, define $G_n(z) = p_0 + p_1 \cdot z + p_2 z^2 + \cdots + p_k \cdot z^k + \cdots$. From the above difference equations for $F(n)$, Metric obtains difference equations for $G(n, z)$, using a transformation discussed below:

$$G(0, z) = z^{c0}$$
$$G(n + 1, z) = a \cdot z^{c1} + (1 - a) \cdot z^{c2} \cdot G(n, z).$$

Treating z as a parameter, this is a simple system in n, having the form:

$$H_0 = d_0$$
$$H_{n+1} = d_1 + d_2 \cdot H_n.$$

Hence, its solution has been discussed previously:

$$G(n, z) = az^{c1}/(1 - (1 - a)z^{c2}) + (1 - a)^n z^{n \cdot c2}(z^{c0} - az^{c1}/(1 - (1 - a)z^{c2})).$$

Since $G(n, z)$ is a probability generating function:

$$mean(G_n) = G'_n(1)$$
$$variance(G_n) = G''_n(1) + G'_n(1) - (G'_n(1))^2.$$

Taking the first and second derivatives of the above expression for $G(n, z)$ and simplifying, Metric obtains the

[6] In the interest of brevity, we refer the reader to other sources for the development of generating functions and their use in program analysis: [17] contains a good introduction to generating functions; [16] discusses the use of generating functions where the program is modeled as a discrete Markov process; and [9] makes extensive use of generating functions in the analysis of algorithms.

desired mean and variance:

$$mean(F(n)) = e_0 + e_1\cdot(1-a)^n$$
$$variance(F(n)) = e_2 + e_3\cdot(1-a)^n + e_4\cdot n(1-a)^n + e_5\cdot(1-a)^{2n}$$

where the e_i's are functions of the c_i's. For example:

$$e_1 = null + 3\cdot vref + fncall + cdr - (null + 5\cdot vref + fncall + 2\cdot cdr + eq)/a.$$

As a final comment on this example, we note the crucial role of algebraic simplification in both computing and presenting the final result. In carrying out algebraic manipulation, it is usually necessary to simplify the result at each stage; otherwise, intermediate expression swell can consume unreasonable amounts of storage. The system, in fact, simplifies the result of every algebraic operation. For example, a sum of terms is represented by an n-ary "bush" in which cancellation has been carried out and, more generally, in which terms differing only by a constant factor have been grouped together. The algebraic *plus* routine constructs this simplified representation in forming its answer. A second role of algebraic simplification is expressing final results in a form which clearly displays the dependence on the parameter(s), e.g. n in the above example. Metric simplifies the final result explicitly to achieve this: let $x_1, \cdots, x_n$ be the parameters. Each term of the result is written in the form $D_i\cdot F_i(x_1, \cdots, x_n)$ where D_i is independent of the x_1's but may depend on free variables. Terms which differ only in D_i are collected together; the result is a sum on j of terms $(D_j^1 + \cdots + D_j^{k[j]})\cdot F_j(x_1, \cdots, x_n)$. The form is then simplified by defining new constants $C_j = (D_j^1 + \cdots + D_j^{k[j]})$.

We next consider the treatment of **when** qualifications in the general case. The performance is computed by obtaining the *range* and the *moments*. Consider obtaining *max*. First, the difference equations are rewritten by replacing any performance subexpressions with their *max* component.[7] Next, a reduced system is formed by eliminating any early exit cases which allow termination short of recursion down to the base case; the solution R to the reduced system is obtained. If there are no early exit cases, then R is the desired maximum. If there are early exits, then the maximum obtained by taking such an exit occurs if it is taken at the last possible recursion step. Hence, the system is next solved under this assumption. The desired *max* is the maximum of the two solutions thus obtained. The *min* is computed analogously. Then the *min* and *max* are compared. If they are equal, a scalar result is returned; otherwise, the moments must be computed to complete the performance expression.

The computation of *moments* is somewhat complex. First, the generating function is obtained. Deriving a difference equation for the generating function is essentially syntactic: on the left hand side of a case definition, $F(n)$ is replaced by $G(n, z)$; on the right hand side of a case definition, the transformation g is applied.

(1) $g(a_1; a_2) = g(a_1) + g(a_2)$
(2) $g(c$ **when** $e) = probability(e)\cdot g(c)$
(3) $g(c1 + c2) = g(c1)\cdot g(c2)$
(4) $g(F(n)) = G(n, z)$
(5) $g(s) = z^s$ if s is a scalar independent of n
(6) $g(r) = R(z)$ if r is a nonscalar performance independent of n, where R is a new function letter.

The first rule establishes that if the right hand side is a set of alternatives, then the transforms of the alternatives are to be summed. The second rule establishes that the generating function for a **when** qualified case is the probability of the event times the generating function for the case.

The last rule brings up a new point. It maps a nonscalar performance r into a function $R(z)$—the generating function for the probability distribution of that performance. R is not explicitly known. However, an explicit representation of R is not really necessary. The mean and variance of G depends on R only through the values of its zeroth, first, and second derivatives evaluated at $z = 1$. Since R is a probability generating function, $R(1) = 1$, $R'(1) = mean(r)$, $R''(1) = variance(r) - mean(r) + mean(r)^2$. The *mean* and *variance* of a performance r are known. Hence, R can be treated as a formal function having these properties. It will be noted that rule (5) is a special case of this rule, since for any scalar S, $mean(S) = S$ while $variance(S) = 0$.

As an example of how the transformation g operates, consider the difference equation for the time of *COUNT*:

$$F(0) = c0$$
$$F(n+1) = c1 + F(n) \textbf{ when } X = s_1;\quad c2 + F(n) \textbf{ when } X \neq s_1.$$

The result of g is a simple difference equation for the generating function:

$$G(0, z) = z^{c0}$$
$$G(n+1, z) = (a\cdot z^{c1} + (1-a)z^{c2})\cdot G(n, z)$$

where $a = probability(X = s_1)$. As a second example, suppose that in *COUNT* the operation *ADD*1 was replaced by some other operation having an execution time described by a nonscalar performance (cf. Section 4.1). In that case, the coefficient corresponding to $c1$ would be a nonscalar performance and, under the transformation g, this would be mapped into a probability distribution $R_1(z)$—using rule (6) above. Hence, the difference equation for the generating function would then be:

$$G(0, z) = z^{c0}$$
$$G(n+1, z) = (a\cdot R_1(z) + (1-a)\cdot z^{c2})\cdot G(n, z).$$

[7] This assumes that all performances can attain their maxima simultaneously. This is another instance of assuming independent tests, and is not always correct. It may be that, due to some coupling, when one module has worst case behavior, some subsequent module cannot. In general, the computed *max* and *min* are upper and lower bounds but are not necessarily attained.

Next, a closed form for the generating function is obtained by applying the difference equation solver to the new difference equation. With a closed form thus produced, obtaining the first and second derivatives is straightforward. If R_i's are present, their first and second derivatives are represented formally. The only complication is the pervasive use of algebraic simplification to control the size of the expressions [13]. Evaluating at $z = 1$ and using the known values of the zeroth, first, and second derivatives of the R_i's yields the desired results.

7. Concluding Remarks

The development of Metric has been concerned with complete automation: mechanical analysis of programs with no assistance. As such, it complements work such as [5] on providing interactive tools for use by the programmer. In its current state, Metric can analyze only fairly simple Lisp programs, whereas an interactive system has the potential for handling programs of arbitrary difficulty. It therefore is appropriate to address the issue of extending this work, i.e. to identify the problems which must be solved in scaling up the system to handle a richer class of programs.

Languages such as Fortran, Algol, or PL/I present a large variety of constructs absent from our simple Lisp subset. However, the treatment of many of these within our framework is basically understood.

(1) Control structure. Well-nested loop constructs (e.g. **do**, **for**, **while**) correspond directly to nested recursive procedure calls.

(2) Side effects. Assignments in straight line code can be modeled by successive substitutions. Assignments around a loop are modeled by the recursion relations they define.

(3) If and case statements. Test and branch statements of all sorts are syntactic variants of conditionals. The *COUNT* example shows the treatment and resulting analysis of loops with embedded **if** statements. The *MEMBER* example shows the treatment of a **for** loop with an exit condition.

(4) Optimization. As noted in Section 2, if the source program is not mapped one-for-one onto the machine then the local cost assignment should be performed after all significant optimization has been performed.

It appears that the most significant problems are more fundamental, having more to do with the theory of computation than with programming languages. The most important is the probabilistic treatment of tests. As noted in Section 4, all tests are currently treated as independent events. This simplifying assumption is often wrong, e.g.:

```
if $x = y$ then ... if $x = y$ ...
if $x < y$ then ... if $y < z$ ...
```

Once detected, repeated identical tests such as the first example can be handled satisfactorily; the probability of the redundant test failing is zero. The problem of detecting simple common cases here is identical to that required for test elision in an optimizing compiler, e.g as discussed in [18]. The more complex situation where the outcome of one test forces the outcome of a subsequent nonidentical test reduces to proving the validity of a logical implication. Domain-specific theorem provers such as those being developed for program verification can be employed here. The difficult problem is cases like the second where the conditional probability of $test_2$ given the success of $test_1$ is neither 0, 1, nor the same as the unconditional probability of $test_2$. Detecting the possibility of conditioning or, equivalently, guaranteeing its absence is fairly straightforward. If the conditional probability is constant, it can be measured. However, the important case where $test_2$ is conditioned and nonconstant is difficult. Mechanization would seem to be beyond the range of current techniques.

A possible prospect may be to proceed by analogy with program verification: to allow the addition to the program of performance specifications by the programmer, which the system then checks for consistency. That is, performance expressions are treated as assertions and the task of the system is to verify that the resource analysis provided by the programmer is correct.

The analogy with verification is further evidenced when we observe that the correct determination of conditional probability is required not only to obtain *mean* and *variance*, but also *max* as well. Consider, for example, the following simple program to sort an array $A[1 : n]$

```
flag ← true;
while flag do
begin flag ← false;
        for i from 2 to n do
        if $A[i - 1] > A[i]$ then
        begin flag ← true; exchange ($A[i - 1]$, $A[i]$)
        end
end
```

Since the outer loop is executed until some pass on which no exchange occurs, termination depends on the test $A[i - 1] > A[i]$ being affected by prior tests and exchanges.

Beyond this, there are a number of defects in the current system whose solutions are understood. The algebraic manipulation subsystem could be augmented with a radical simplification package [13]. Similarly, the methods used for solution of difference equations could be extended. Also, the current organization into phases is only a linear approximation to the right one: currently, the source program is transformed in successive phases until an answer is obtained; however, guesses are made along the way and if certain of these are wrong, the system fails. An obvious improvement would be a more flexible organization where a later phase can report back reasons for failure, earlier stages can ask for advice from later ones, and several approaches can be tried in parallel. Another area for improvement is the final representation of analyses: algebraic expressions are sometimes advantageously presented by approximating an exact but complex solution. Some facilities for dealing with approximations are therefore desirable.

Acknowledgments. We have benefited greatly from discussions with L.P. Deutsch, L. Guibas, and J Moore.

Received July 1974; revised November 1974

References

1. Beizer, B. Analytical techniques for the statistical evaluation of program running time. Proc. AFIPS 1970 FJCC, Vol. 37, AFIPS Press, Montvale, N.J., pp. 519–524.
2. Bobrow, D., and Murphy, D. Structure of a Lisp system using two level storage. *Comm. ACM 10*, 3 (Mar. 1967), 155–159.
3. Cheatham, T.E., and Wegbreit, B. A laboratory for the study of automating programming. Proc. AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 11–21.
4. Cocke, J., and Schwartz, J.T. *Programming Languages and Their Compilers.* Courant Institute of Mathematical Sciences, New York U. Press, New York, 1970.
5. Cohen, J., and Zuckerman, C. Two languages for estimating program efficiency. *Comm. ACM 17*, 6 (June 1974), 301–308.
6. Green, C. Application of theorem-proving to problem solving. Proc. First Internat. Joint Conf. on Artif. Intell., 1969, pp. 219–239.
7. Ingalls, D. The execution time profile as a programming tool. In *Design and Optimization of Compilers*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 107–128.
8. Knuth, D.E. An empirical study of FORTRAN programs. In *Software-Practice and Experience*, Vol. 1, 1971, pp. 105–133.
9. Knuth, D.E. *The Art of Computer Programming.* Addison-Wesley, Menlo Park, Calif., 1968.
10. Lee, R.C.T., Chang, C.L., and Waldinger, R.J. An improved program-synthesizing algorithm and its correctness. *Comm. ACM 17*, 4 (Apr. 1974), 211–217.
11. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM 3*, 4 (Apr. 1960), 184–195.
12. Manna, Z., and Waldinger, R.J. Toward automatic program synthesis. *Comm. ACM 14*, 3 (Mar. 1971), 151–164.
13. Moses, J. Algebraic simplification: A guide for the perplexed. *Comm. ACM 14*, 8 (Aug. 1971), 527–537.
14. Nemeth, A.G., and Rovner, P.D. User program measurement in a time-shared environment. *Comm. ACM 14*, 10 (Oct. 1971), 661–666.
15. Proceedings of a symposium on very high level languages. SIGPLAN Notices, Vol. 9, No. 4 (Apr. 1974).
16. Ramamoorthy, C.V. Discrete Markov analysis of computer programs. ACM 20th Nat. Conf., 1965, pp. 386–392.
17. Riordan, J. *An Introduction to Combinatorial Analysis.* John Wiley, New York, 1958.
18. Sites, R.L. Proving that computer programs terminate cleanly. Ph.D. Th., Comput. Sci. Dep., Stanford U., May 1974.
19. Teitelman, W. *Interlisp Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, Calif., 1974.
20. Weissman, C. *Lisp 1.5 Primer.* Dickenson Pub. Co., Belmont, Calif., 1967.

PROFESSIONAL ACTIVITIES

Calls for Papers: Important Dates

15 September 1975. 1975-76 ACM George E. Forsythe Student Paper Competition. See April Communications. Any student who has not received a bachelor's degree before April 1, 1975, is eligible. Submit intent by June 30 and complete paper by September 15 to ACM Student Editorial Committee, Department of Computer Science, University of Toronto, Toronto, Ont., Canada M5S 1A7.

1 October 1975. 3rd International Colloquium on Automata, Languages, and Programming, Edinburgh University, Scotland, July 20-23, 1976. See August *Communications*. Submit intent and topic by October 1 and draft paper (1,000-10,000 words in four copies) by November 15 to C.A. Mackinder, Organizing Sec., Computer Science Dep., Edinburgh University, James Clark Maxwell Bldg., Mayfield Road, Edinburgh, Scotland.

1 October 1975. Conference on Data: Abstraction, Definition and Structure, Salt Lake City, Utah, March 22-24, 1976. Sponsors: ACM SIGPLAN and SIGMOD. See April Communications. Send five copies of complete papers with abstracts to Prog. Chm: Henry F. Ledgard, Computer and Information Science, Graduate Research Center, University of Massachusetts, Amherst, MA 01002.

1 October 1975. Computer Society of India Annual Convention on "Computers and Social Change," Hyderabad, India, January 20-23, 1976. Write Program Chairman (DVR Vithal, Computer Group, Tata Institute of Fundamental Research, Bombay 400 005, India) for information for authors, giving title and subject, immediately; abstracts due **October 1;** full papers due **November 1.**

10 October 1975. COMPCON 76, Jack Tar Hotel, San Francisco, Calif., Feb. 24-26, 1976. Sponsor: IEEE-CS. See September *Communications*. Abstract and 1000-2000 word digest due **October 10,** to Herschel H Loomis Jr., Dep. of Electrical Engineering, University of California, Davis, CA 95616.

15 October 1975. 8th AICA Congress on Simulation of Systems, Delft, The Netherlands, August 23-28, 1976. See August *Communications*. Submit two copies of 250-500 word abstract in English to L. Dekker, Delft University of Technology, Department of Mathematics, Julianalaan 132, Delft/The Netherlands. Notifications by December 31; provisional full paper due February 29, 1976; notifications by April 30; final paper due June 30. Proceedings.

26 October 1975. Technical Symposium on Computer Science and Education, Disneyland Hotel, Anaheim, Calif., February 12-13, 1976. Sponsors: ACM SIGCSE and SIGCUE. Send three copies of complete paper to Ronald W. Colman, Computer Science Programs, California State University, Fullerton, CA 92634.

1 November 1975. Fifth Biennial International Codata Conference, University of Colorado, Boulder, Colo. Sponsor: The Committee on Data for Science and Technology (CODATA), ICSU. See August *Communications*. Submit title and brief description to Program Committee Chm: David R. Lide Jr., NBS, Washington, DC 20234; notifications and instructions by January 1.

1 November 1975. 1976 International Symposium on Fault-Tolerant Computing (FTCS-6), Pittsburgh, Penna., June 21-23, 1976. Sponsor: IEEE-CS TC on Fault-Tolerant Computing. See September *Communications*. Send abstract (250-word max.) to Program Chairman: Barry R. Borgerson, Sperry Research Center, 100 North Road, Sudbury, MA 01776. 4 c. manuscript (4000-word max.) and 1-p. statement on paper's contribution due **December 1;** notifications by March 1, 1976.

10 November 1975. Programming Systems in the Small Processor Environment, New Orleans, Louisiana, March 4-6, 1976. Sponsors: ACM SIGMINI and SIGPLAN. See August *Communications*. Five copies of manuscript with abstract, not to exceed 16 double-spaced typewritten pages, to Program Chairman: Lawrence J. Schutte, Room 6B-302, Bell Telephone Laboratories, Naperville, IL 60540; 312 690-4116. Notifications by December 10; final papers due **January 15, 1976.** Proceedings.

15 November 1975. 1976 International Symposium on Information Theory, Ronneby Brunn, Ronneby, Sweden, June 21-24, 1976. Sponsor: IEEE Information Theory Group. See August *Communications*. Submit complete manuscript and abstract for "long" paper (30 min duration) and 500-word summary and abstract for "short" paper (15 min duration) to Jack Salz, Bell Laboratories, Room 1G-509, Holmdel, NJ 07733. Notifications by Feb. 1.

15 November 1975. 2nd International Symposium on Programming, Paris, France, April 13-15, 1976. Sponsors: Centre National de la Recherche Scientifique (CNRS) and Universite Pierre et Marie Curie. Submit an abstract (one page) and first draft of paper (about 12 pages) to Secretariat du Colloque, Institut de Programmation, 4, Place Jussieu, 75230 Paris, Cedex 05, France. Notifications by January 15, 1976; final papers due by **March 1, 1976.** Proceedings.

1 December 1975. Fourth Annual Computer Science Conference, Disneyland Hotel, Anaheim, Calif., Feb. 10-12, 1976. Sponsor: ACM. See August *Communications*. Submit abstract of research report to Chairman: Julian Feldman, Dep. of Information and Computer Science, University of California, Irvine, CA 92664.

1 December 1975. 1976 Summer Simulation Conference, Sheraton-Park Hotel, Washington, D.C., July 12-14, 1976. Sponsors: AICHE, AIAA, AMS, ISA, SCI, and SHARE. See July *Communications*. Send three-to-five page summary to Program Chairman, Iwao Sugai, JHU Applied Physics Laboratory, 11100 Johns Hopkins Road, Laurel, MD 20810. Notifications by Feb. 1, 1976; complete manuscripts due March 15. Proceedings.

1 December 1975. Third ICASE Conference on Scientific Computing: Computer Science and Scientific Computing, Quality Inn/Fort Magruder, Williamsburg, VA, April 1-2, 1976. Sponsor: ICASE in coop. with ACM, ACS, AIAA, ASCE, IEEE, SIAM. Submit 2-3 page abstracts for papers to be presented in poster sessions to Robert G. Voigt, ICASE, MS-132C, NASA Langley Research Center, Hampton, VA 23665; 804 827-2513.

1 December 1975. International Conference on Computational Linguistics, University of Ottawa, Ottawa, Canada, June 28-July 2, 1976. Submit 1000-word abstract to M. Kay, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94305; text of accepted papers due **May 1, 1976.**

5 January 1976. '76 NCC, New York, N.Y., June 7-10, 1976. See June *Communications*. Submit six copies of paper not to exceed 5,000 words and abstract of not over 200 words to Program Chairman, Stanley Winkler, IBM Corp., 18100 Frederick Pike, Gaithersburg, MD 20760. Notifications by March 1, 1976.

16 January 1976. Conference on Information Sciences and Systems, The Johns Hopkins University, Baltimore, MD, March 31, April 1 and 2, 1976. See September *Communications*. Submit a "regular" or "short" designation, title, and summary to 1976 CISS, Department of Electrical Engineering, The Johns Hopkins University, Baltimore, MD 21218. Notifications by February 16. Proceedings.

1 March 1976. SIGCSE 76, Quality Inn/Fort Magruder, Williamsburg, Va., July 26-27. See July *Communications*. Submit three copies of paper to William Poole, Mathematics Department, College of William and Mary, Williamsburg, VA 23185.