Knowledge-Based Programming: A Survey of Program Design and Construction Techniques

ALLEN T. GOLDBERG

Abstract—**Software development is a knowledge-intensive activity. A promising approach to improving the efficiency of software development is the construction of a knowledge-based software assistant [4]. Such a system provides the medium of interaction for the development process and enforces the semantic consistency of the program as it evolves from its specification to its implementation. A system that provides automated assistance for program development can do so to the extent to which it embodies knowledge of the programming process and knowledge of the application domain for which the program is written. Programmers have knowledge of how to represent problem-domain objects with the data structure facilities of their programming language, how to implement various search techniques and other programming cliches [49], the efficiency of various program constructions, and so on. This paper surveys our understanding and formalization of this knowledge. It is presented from the point of view of algorithm design and high-level program optimization. It discusses techniques for data structure selection, the procedural representation of logic assertions, store-versus-compute, finite differencing, loop fusion, and algorithm design methods.**

Index Terms—**Knowledge-based software development, program optimization, program synthesis, program transformation, very-high-level languages.**

I. Introduction

THIS paper surveys an application of artificial intelligence (AI) to the development of software, specifically the construction of efficient implementations of programs from formal high-level specifications. Central to this discussion is the notion of *program development by means of program transformation*. Using this methodology, a formal specification is compiled (either manually or automatically) into an efficient implementation by the repeated application of correctness-preserving, source-to-source transformations. We use the term *program* to describe any formal description, declarative or procedural, finitely executable or not, that has a precise meaning in a computational sense and *program development* to refer to activities throughout the software lifecycle which modify programs. *Specifications* are just the programs the system starts with, *implementations* are the result. Our use of the term specification does not imply a declarative and/or nonexecutable program. The difference between a specification and an implementation is a matter of degree as to how problem-oriented as opposed to machine-oriented the program is. Program development is just one facet of software development, albeit a fundamental one. AI techniques are applicable to other facets of software development, such as requirements analysis, debugging, testing, and project management. Over the short term, improved support of these activities will make significant contributions to increasing software productivity. This paper restricts its attention specifically to program development, an area for which we expect the benefits to be most significant over the long term.

A. Knowledge-Based Programming

Software development is a knowledge-intensive activity. A promising approach to improving the efficiency of software development is the construction of a knowledge-based software assistant [4]. Such a system provides the medium of interaction for the development process and enforces the semantic consistency of the program as it evolves from its specification to its implementation. A system that provides automated assistance to program development can do so to the extent to which it embodies knowledge of the programming process and knowledge of the application domain for which the program is written. Programmers have knowledge of how to represent problem domain objects with the data structure facilities of their programming language, how to implement various search techniques and other programming cliches [49], the efficiency of various program constructions, and so on. This paper surveys our understanding and formalization of this knowledge. It is presented from the point of view of algorithm design and high-level program optimization. It discusses techniques for data structure selection, the procedural representation of logic assertions, store-versus-compute, finite differencing, loop fusion, and algorithm design methods.

A knowledge-based software assistant will increase productivity by providing a rapid prototyping capability, by reducing the cost of development and maintenance, and by increasing reliability.

Rapid prototyping is the creation of an executable, but perhaps very inefficient, implementation (the implementation may be the specification itself) from a formal specification. Rapid prototyping allows the functionality of the

Manuscript received September 30, 1985. This work was supported by the Defense Advanced Research Projects Agency under Contracts N00014-79-C-0127, N00014-80-C-0045, and N00014-81-C-0582 monitored by the Office of Naval Research (ONR), and by ONR under Contract N00014-84-C-0473.

The author is with Kestrel Institute, Palo Alto, CA 94304 and the Department of Computer and Information Sciences University of California, Santa Cruz, CA 95064.

IEEE Log Number 8608187.

system to be observed before development has occurred, significantly reducing maintenance performed to modify functionality after coding has occurred. Rapid prototyping helps identify code bottlenecks—the critical regions of code that must be carefully optimized. Often infrequently executed code is over-optimized, wasting effort and making maintenance more difficult.

A knowledge-based software assistant can make design and coding more productive by performing the clerical tasks that follow a design decision. For example, if a data structure representation for an object is chosen, the operations performed on the object may be refined into the corresponding operations on the representation automatically. A formal description of the derivation of an implementation, called the *development history* is constructed as the development proceeds. The development history records the application of each transformation, perhaps with a justification of the implementation decision associated with the transformation.

Maintenance is performed on the specification rather than by performing difficult surgery on the implementation. Specifications are easier to modify than implementations for three reasons. First, design decisions may appear implicitly in the implementation and may not be properly documented. Hence a modification to an implementation may inadvertently violate implicit constraints on the code implied by the design. Second the process of implementation is one of *information spreading*, that is, assertions describing the problem simply expressed in the specification may be reflected diffusely throughout the implementation. Modifications can violate these assertions. Third, the implementation is cluttered with knowledge about performance on the target architecture whereas the specification deals primarily with knowledge about the problem domain. The implementation of the modification is achieved by *replay* of the previous development, which must be altered only to reflect the modification. For example, an assertion in the specification may constrain a variable to represent the length of a list. The assertion may, for example, be maintained by updating the value of the variable whenever the list is modified. A modification performed on the implementation may alter the list and the programmer may fail to update the variable. But if the modification is made to specification, when the development is replayed, the update to the variable will be generated.

Using a knowledge-based software assistant, reliability is enhanced. Because the transformations are correctness-preserving, the implementation is equivalent to the specification. Proving a transformation correctness-preserving is nontrivial, but this is only done once. If its correctness is not formally established, extensive use of the transformation improves our confidence in its correctness just as extensive use of a compiler improves our confidence in its correctness.

More detailed and elegant discussions about formal program development can be found in [39], [42], and [4].

B. *Expert Systems and Compilers*

A knowledge-based software assistant many be implemented using AI expert system technology. The basic components of an expert system are a knowledge-base, a set of production rules, whose application affects the contents of the knowledge-base, and a control strategy for selecting and applying rules. Programs may be represented by their abstract syntax trees in the knowledge-base. Program transformations may be represented as production rules. The fact that there are many possible implementations of a programming construct means that at any given time many rules, each reflecting a different implementation choice, may be applied. By placing the specification at the root of a tree and by recursively considering all possible transformations that may be applied to each node of the tree, the space of equivalent programs that the system may derive is (implicitly) generated. The control strategy must navigate this tree to find an efficient implementation of the specification. User guidance and heuristic search based on an efficiency analysis may guide the search.

Knowledge-bases provide a general and uniform mechanism for representing data. This contrasts with a compiler in which different data structures are used for representing the abstract syntax tree, the symbol table, intermediate code, etc. The uniform access provided by the knowledge-base permits integration of diverse tools that may be used during transformation, such as algebraic simplifiers, theorem provers, and program analysis tools. Furthermore, information about the externals of the program, such as project-management data, documentation, and testing data can be stored in the knowledge-base. Using a knowledge-base adds flexibility while sacrificing the efficiency that more specialized data structures offer.

Functionally, a program transformation system and a compiler are identical, both translate a high-level program (specification) to a lower-level, semantically equivalent implementation. The development of efficient compilers has been a central success of computer science. Why is it desirable to employ a new methodology, that of program transformation? Compilers, relying on syntax-directed translation are well suited for translation tasks in which each source construct has as a single, natural implementation in the object language. Thus a primarily local approach of using the syntax to recursively decompose a program into elementary constructs that can be directly translated suffices. If any global analysis is done, it is in an after-the-fact optimization phase. This strategy is awkward to use if the language contains constructs which have many possible implementations with varying efficiency characteristics and for which selection of the proper implementation requires global analysis. Thus the possibility of interaction with the user and the flexibility and modifiability of transformation systems is preferred over the efficiency of a compiler.

For example, sets can be implemented as hash tables, lists, arrays, trees, bit-vectors, property lists, etc. The

proper implementation of a set-valued object requires a global analysis of the operations performed on the set, its relation to other objects in the program, its size, the type of its elements, etc. Sets as they appear in Pascal are severely restricted (elements must be a subset of a small enumerated type) so that their implementation as a bit-vector is efficient in all possible contexts. The translation of high-level specifications, containing constructs with no single, tolerably efficient implementation benefits from a more flexible organizing principle than syntax-directed translation. The transformation paradigm provides that flexibility.

Indeed technology developed for compilers—parsers, parser generators, pretty printers, type checkers, and the like—are essential components of knowledge-based assistants. Optimization and program analysis techniques need to be expanded and improved. Especially useful are *incremental analysis* tools [37] which efficiently update program-analysis data when a program undergoes a change as the result of the application of a transformation.

C. Language and Transformation

Program development by transformation is a methodology appropriate largely independent of programming language. Transformational developments in the literature often utilize logic or functional languages. These languages have often been proposed for the next generation of very-high-level languages and many problems have elegant expression within them. Program transformation systems do impose an important requirement on language. Because a transformation system performs translation in small steps, each step the application of a single transformation, it is best to envision the translation as occuring within the context of a single *wide-spectrum language* containing constructs used both for specification and implementation. The transformations are then *source-to-source* transformations within a single semantic framework.

An idea important to this paper is the categorization of transformations by their *granularity*, which is the degree to which they alter the structure of the program. Low-granularity transformations make small, local changes to the program. A transformation which applies an algebraic law (such as communitivity) to an expression is an example of a low-granularity transformation. High-granularity transformations make dramatic, global changes to a program. Given a specification in a prealgorithmic form, a transformation which instantiates a divide-and-conquer schema to the specification is an example of a high-granularity transformation. A transformation expressing a data structure choice is medium grain.

A useful analogy can be made between the structure of mathematical proofs and program developments. The conventional presentation of a mathematical proof is structured by lemmas and case analysis. Theorems are invoked. A mathematical proof can be reduced to a long sequence of low-level inferences, but in doing so its structure and intuition is lost. In a program development the structure and intuition of the development is provided by the use of high-granularity transformations. The high-granularity transformations express algorithm design and optimization techniques that have been demonstrated to have wide application. Significant program derivations have been presented in the literature using only low-granularity transformations. These derivations tend to be long and unintuitive. A robust program development system must include both high- and low-granularity transformations to be a practical tool. Low-granularity transformations are used to massage a program into a form for which a high-granularity transformation applies. Balzer [3] refers to this massaging as *jittering*. Mathematical proofs have similar characteristics. Manipulations, that can be called jittering, are used to bring the problem into a form in which a powerful theorem, analogous to a high-granularity transformation, apply. From the point of view of AI, there is little knowledge implicit in low-granularity transformations and a system relying only on them must have a sophisticated control strategy to be effective.

The transformational approach promises attractive solutions to most of the problems plaguing software development, but to date there has been no system created which demonstrates that, for a reasonably general class of problems, software can be produced more cheaply using this rather than traditional methods. However, transformational systems have been effectively used to solve certain special compilation problems. Boyle and Muralidharan [9] have used a transformation system as part of a cross compiler that translates Lisp code to Fortran. Haradhvala, Knobe, and Rubin [23] have used a transformation system as part of the back-end of a compiler to translate intermediate code into highly optimized machine code. The CHI system [46] rapid prototypes a very-high-level language, based on logic and set-theoretic data-types, into Lisp using a transformation system. All of these systems work automatically and produce high-quality code. The broader goal of the CHI project is the construction of a knowledge-based programming environment. Many of the techniques described here are being incorporated into the CHI system. There are quite a number of program transformation systems under development. It is not our goal to survey each system; the interested reader is referred to [35]. This paper is organized as follows. In Section II program transformation is looked at in the context of logic programming. In Section III we consider the transformational development of functional programs. Sections II and III assume some knowledge of logic and functional programming, respectively, such as that which may be found in [28] and [13]. In Section IV we report on the extent to which programming knowledge has been formalized. Section V gives some conclusions.

II. The Logic Approach to Program Development

A logic program is simply a collection of formulas of the first-order predicate calculus. These formulas state

facts and properties about the problem domain. Ideally the construction of a logic program proceeds by formalizing these facts without regard to how they will be used computationally. For example consider the set of assertions:

$\forall w[\textit{simple-list}(w) \leftrightarrow$

$(w=\textbf{nil} \vee \exists x \exists y(w=x.y \;\&\; \textit{element}(x) \;\&\; \textit{simple-list}(y)))]$

$\forall x \forall y[\neg(x.y = \textbf{nil})]$

$\forall x \forall y \forall u \forall v[x.y=u.v \leftrightarrow (x=u \;\&\; y=v)]$

$\forall x \forall w[x \in w \leftrightarrow (\textit{element}(x) \;\&\; \textit{simple-list}(w) \;\&$

$\exists y \exists v[w=y.v \;\&\; (x=y \vee x \in v)])]$

These assertions comprise an axiomatization of the datatype *simple-lists*, lists which are either empty or constructed from a *simple-list* by prepending an element in front. The empty list is denoted by the constant ***nil*** and the prepend operation by the function ".". The datatype *simple-list* is represented by a predicate of the same name which is true precisely for those objects comprising the datatype. *element*(x) is a predicate true for those objects which may appear as elements of a *simple-list*. The axiomatization of *element* is not given. The membership predicate, $x \in w$, is true if x is an element in *simple-list* w. The program contains an assertion, given below, axiomatizing a relation on *simple-lists* called *less-all*(x, w, t). *less-all*(x, w, t) is true if x is an element, w a *simple-list*, and x is less than all elements on list w and t is 1 or x is not less than all elements on w and t is 0. It is assumed that less than ($<$) has been defined as a predicate on the datatype *element*. The axiom is

$\forall w \forall x \forall t[\textit{less-all}(x, w, t) \leftrightarrow$

$(\textit{element}(x) \;\&\; \textit{simple-list}(w) \;\&$

$((\forall u\ (u \in w \rightarrow x<u\ ) \;\&\; t=1) \vee$

$(\neg \forall u(u \in w \rightarrow x<u) \;\&\; t=0))]$

These assertions are used as axioms from which new assertions are proved. The formula $\exists t[\textit{less-all}(2,\ 3.4.5.\textbf{nil},\ t)]$ is provable from the above axioms. Constructive proofs of assertions like this one are the computational basis of logic programming. A constructive proof not only establishes the validity of the assertion but provides an instantiation of variables that appear existentially bound. In this case a constructive proof instantiates t to the value 1. The effect of the proof is to compute that 2 is less than all members of the list $3.4.5.\textbf{nil}$.

The relation *less-all* defines the input–output relation computed by this logic program. It does so without specifying a *functional use* of the program, i.e., a commitment to which variables are the inputs and which are the outputs. Proving the assertion $\exists t[\textit{less-all}(2,\ 3.4.5.\textbf{nil},\ t)]$ specifies one functional use of the program, the assertion $\exists w[\textit{less-all}(2,\ w,\ 1)]$ specifies another (which is to construct a list, all of whose element are greater than 2). The formula $\forall w \exists y\ [\textit{less-all}(y,\ x,\ 1)]$ asserts that for any *simple-list* w there is a y which is less than all the elements on the list w. This assertion will be true if $<$ is a total order with no minimal element on the datatype *element*. Assume that this is the case. In this formula the existentially quantified variable y depends on the universally quantified w. Thus a constructive proof of the assertion constructs a computable function which takes as argument a *simple-list* w and returns a value y which is less than all the elements on the list w. The ability of logic programming systems to return symbolic results, i.e., expressions with free variables, is an important capability of these systems. It obscures the distinction between program construction and execution. In this last example, the result of the computation was a program.

The notion of using a constructive proof to synthesize a program was the basis of early work in automatic programming [20], [48]. In this work a functional program is derived from a logic specification of the form $\forall y_1 \cdots \forall y_n \exists x\ Q(x, y_1, \cdots, y_n)$. This is a natural form of specification when the y's are regarded as inputs and the x as output. A constructive proof of the formula is used to extract a (possibly recursive) functional program which computes x given as input the y's. Green introduced the idea of recording the substitutions of terms for variables made (via unification) in the course of a resolution proof. This is the exclusive mechanism which Prolog uses to construct output.

Manna and Waldinger [29] have continued research on the deductive approach to program synthesis. Their emphasis is on automatic synthesis rather than semiautomatic transformation. They use a deductive method based on a nonclausal resolution. The proof system supports case-analysis that introduces conditionals and well-founded induction that introduces recursive functions into programs. The system uses control strategies to direct the search for a constructive proof. The *polarity strategy* restricts the application of the resolution rule to those instances likely to generate valuable conclusions. The *recurrence strategy* matches goals with subgoals to discover useful recursive definitions. In [30] Manna and Waldinger present a deductive synthesis of the unification algorithm.

A model of program execution based on deduction in the full first-order predicate calculus is severely limited because of the general ineffectiveness of automatic theorem provers. However if the assertions of the program are restricted to be Horn Formulas and the program written with some understanding of how the theorem prover will execute, then computation based on deduction becomes viable. A Horn formula is one in the form $P_1 \;\&\; P_2 \;\&\; \cdots \;\&\; P_n \rightarrow Q$, with $P_1, \cdots, P_n, Q$ atomic formulas. This is just the idea exploited in Prolog [28]. An interpreter for Prolog is a Horn formula resolution theorem prover utilizing backward inference and depth-first search.

This suggests the approach of starting with a specification written in the full first-order predicate calculus and deriving by transformation an implementation in the re-

stricted language of Horn formulas. The resulting implementation is run as a Prolog program. This is the approach taken by the UPMAIL group (Uppsala Programming Methodology and AI Laboratory). Because the specification and the implementation are both logic programs the transformation rules are simply deduction rules. But since the deduction is no longer carried out automatically, a natural deduction system is used. Natural deduction proof systems are oriented to support deduction by humans. Resolution is a machine-oriented proof system. Given the logic program for *less-all* the development proceeds by deducing the following three Horn formulas as theorems derived from the original logic program:

$$\textit{less-all}(x, \textbf{nil}, 1) \leftarrow \textit{element}(x)$$

$$\textit{less-all}(x, y.z, 0) \leftarrow \textit{element}(y) \;\&\; x \geq y \;\&\; \textit{simple-list}(y.z)$$

$$\textit{less-all}(x, y.z, t) \leftarrow \textit{element}(y) \;\&\; x < y \;\&\; \textit{less-all}(x, z, t)$$

Here the implication $A \rightarrow B$ is written $B \leftarrow A$ as is customary for Horn formulas. These theorems together constitute a Prolog program for computing the *less-all* relation.

We omit the derivation of this program. It requires over 100 deductive steps. Thus the intuition of the derivation is lost in the technical details. The difficulty is the low granularity of the transformations.

The assertions *element*(y) and *simple-list*($y.z$) are assertions expressing type information. The uniform treatment of type information and other assertions reduces clarity and implies expensive runtime checking. Recognition of the distinction between type assertions and other assertions allows type checking and type inference to be done using specialized procedures by the development system automatically. This kind of capability is characteristic of a higher-granularity approach—the system contains knowledge about type that helps support the activity of developing an efficient implementation.

The Horn formula implementation of *less-all* has an intended functional use that regards the element x and the list w as input and t as output. The implementation performs this computation by a linear, front-to-back traversal of the list w. The axiomatization in the specification of *less-all* is nonrecursive, in particular it does not assert that x is less than all elements on list w if x is less than the first element of w and *less-all* is true for x and the remainder of w. The derivation starts from this nonrecursive characterization of *less-all* and derives a linear search implementation. The linear recursive structure of the algorithm was "lifted" from the linear recursive structure implicit in the axiomatization of the data type *simple-list*. In this sense the derivation was data-structure driven. This is a common technique in algorithm synthesis.

Suppose we wished to derive an implementation of *less-all* which divides the list in two halves and recursively computes the *less-all* relation on each half. Such an algorithm would be desirable for implementation on a parallel machine which can work concurrently on the two halves. This would be achievable by including in the axiomatic description of *simple-lists* the operation concatenation. Lifting this operation, i.e., deducing how the relation *less-all* distributes over concatenation would be the key step in the derivation.

Both these implementations can be seen as an application of the divide-and-conquer schema discussed in Section IV. Using a high-granularity approach to program development the derivation would be structured as an application of this schema.

Using the deductive approach, the partial correctness of the implementation with respect to the specification follows from the soundness of the deduction system. A deductive system is *sound* if its inference rules do not permit the deduction of a false conclusion from true premises. *Partial correctness* guarantees that *if* an output is computed by the implementation, the output is correct. This methodology does not guarantee the *total correctness* of the program, that is, whether the implementation will deduce an output when an output is deducible from the specification. For example, if one of the formulas of the implementation of the *less-all* program were dropped the program would be partially correct, but would not return answers in many cases. Furthermore Prolog uses an incomplete theorem prover, i.e., Prolog may fail to deduce conclusions that follow from the Prolog program. The total correctness must be established by an independent proof. These proofs may be specialized to a particular functional use of the program. It can be proved that the Prolog program above will terminate for its intended functional use. The proof is based on well-founded induction over the datatype *simple-list*.

The *less-all* example was taken from [22] which contains developments of more sophisticated algorithms such as *quicksort*. The specification characterizes sorting as finding an ordered permutation of input list and derives a Prolog program for the quicksort algorithm. Eriksson [16] presents a derivation of the unification algorithm for first-order logic in the same formalism.

III. The Functional Programming Approach to Program Development

Functional programming languages provide an excellent medium for the investigation of program improvement by transformation because of their simple semantics and naturalness of expression. A set of fundamental, low-granularity transformations were introduced by Burstall and Darlington [11]. Many transformational developments found in the literature employ the fold-unfold methodology based on this set of transformations. Following [11], a specification is a functional program written as a list of recursion equations. The implementation is another functional program with improved efficiency characteristics but less clarity. Consider the specification:

$$fact(0) \leftarrow 1 \quad (1)$$

$$fact(n+1) \leftarrow (n+1) * fact(n) \quad (2)$$

$$factlist(0) \leftarrow \textbf{nil} \quad (3)$$

$$factlist(n+1) \leftarrow cons(fact(n+1), factlist(n)). \quad (4)$$

This is a specification of a program *factlist*(n) that computes the list of the factorials of the numbers from n down to 1. The base and recursive cases of a recursively defined function are written as separate equations. The recursion equations in lines (1) and (2) serve as a definition of the function *fact*. Only certain function and constant symbols called *constructors* are allowed to appear in the parameter list on the left-hand side of a recursion equation. The constructor symbols for the natural number are 0 and successor (+1). The constructor symbols for lists are ***nil*** and *cons*. This restriction is imposed so that the computation specified by the equations is deterministic.

This definition of *factlist* is a natural specification of the problem. It factors, according to principles of stepwise refinement, the problem of computing the factorial function from the problem of constructing the list of factorials. The difficulty is that the program is inefficient. It performs $O(n^2)$ multiplications when only $O(n)$ are needed. The source of the inefficiency is that the computation of $fact(n+1)$ recomputes $fact(n)$ which had been computed as the previously constructed element of the list. We wish to derive a program that avoids this inefficiency.

A *substitution instance* of an equation is an equation obtained by substituting a term built from constructors for a variable appearing in the equation. If E is an expression then $E[u_1/F_1, \cdots, u_n/F_n]$ denotes the expression in which u_i is substituted for F_i.

The derivation will use the following transformation rules.

Definition: Introduce a new recursion equation whose left-hand side is not an instance of an equation which already appears.

Instantiation: Introduces a substitution instance of an existing equation.

Unfolding: If $E \leftarrow E'$ and $F \leftarrow F'$ are equations and there is an occurrence in F' of an instance of E, replace it with the corresponding instance E', obtaining F''. Add the equation $F \leftarrow F''$.

Folding: If $E \leftarrow E'$ and $F \leftarrow F'$ are equations and there is an occurrence in F' of an instance of E', replace it with the corresponding instance E, obtaining F''. Add the equation $F \leftarrow F''$.

Abstraction: We may introduce a ***where*** clause by deriving from a previous equation $E \leftarrow E'$ a new equation,

$$E \leftarrow E'[u_1/F_1, \cdots, u_n/F_n] \textbf{ where } <u_1, \cdots, u_n> = <F_1, \cdots, F_n>.$$

Laws: We may transform an equation by applying any algebraic laws satisfied by the primitive functions appearing on the right-hand side of an equation.

Unfolding replaces a function call by its body, folding replaces the body by the call. Abstraction provides the capability of giving a variable name to an expression. The pointed brackets $< \cdots >$ are used to denote tuples. Functions may return tuples as values. Tuples with two components are called *pairs*.

These rules are applied to the above specification to improve the problem as follows:

$$G(n) \leftarrow <fact(n+1), factlist(n)>. \quad (5)$$

In (5) we have defined a new function. The value of the function $G(n)$ is a *tuple* consisting of two values: $fact(n+1)$ and $factlist(n)$.

$$G(0) \leftarrow <1, factlist(0)> \quad (6)$$

$$G(n+1) \leftarrow <fact(n+2), factlist(n+1)>. \quad (7)$$

Equations (6) and (7) instantiate this new definition by substituting the term 0 and the term $n+1$ for n, respectively. To obtain (6), $fact(1)$ was unfolded to get $1*fact(0)$, $1*fact(0)$ was unfolded to 1 *1 which was simplified using algebraic laws to 1. In (7) the well-known identity $1+1=2$ was applied.

$$G(n+1) \leftarrow <(n+2) * fact(n+1), cons(fact(n+1), factlist(n))>. \quad (8)$$

In (8) we have unfolded the definitions of both *fact* and *factlist* into the left-hand side of (7).

$$G(n+1) \leftarrow <(n+2) * u, cons(u,v)> \textbf{ where } <u,v> = <fact(n+1), factlist(n)> \quad (9)$$

Equation (9) introduces a ***where*** abstraction. The common subexpression $fact(n+1)$ appearing twice in (8) is abstracted as the value of u. The goal of detecting this common subexpression motivated the definition of G.

$$G(n+1) \leftarrow <(n+2) * u, cons(u,v)> \textbf{ where } <u,v> = G(n) \quad (10)$$

The expression $< fact(n+1), factlist(n)>$ is the body of the definition of the function G. Equation (10) results from folding $G(n)$ for $< fact(n+1), factlist(n)>$.

$$factlist(n+1) \leftarrow cons(u,v) \textbf{ where } <u, v> = < fact(n+1), factlist(n)>. \quad (11)$$

Equation (11) is a ***where*** abstraction of (4).

$$factlist(n+1) \leftarrow cons(u,v) \textbf{ where } <u, v> = G(n). \quad (12)$$

The derivation is completed in (12) where again $< fact(n+1), factlist(n)>$ is folded to $G(n)$. The final program is

$$factlist(0) \leftarrow \textbf{nil}$$

$$factlist(n+1) \leftarrow cons(u,v) \textbf{ where } <u, v> = G(n)$$

$$G(0) \leftarrow \langle 1, factlist(0)\rangle$$

$$G(n+1) \leftarrow \langle (n+2) * u, cons(u,v)\rangle \textbf{ where } \langle u,v\rangle = G(n).$$

This program is an efficient $O(n)$ implementation of the *factlist* program. The common subexpression has been removed and replaced by a ***where*** abstraction.

This example illustrates an obvious but nonetheless important observation, namely that optimization is the exploitation of context to effect program improvement. In this program we exploited the context in which *fact* was computed to speed up its computation. Stepwise refinement facilitates the creation of specifications by decomposing a problem into subproblems and then solving the subproblems independently of context in which they arose. Optimization of the specification results from considering a program fragment in its context of use. The fold-unfold methodology is effective because it captures just this idea. By unfolding the definition of a function, its body can interact with its context by the application of algebraic laws or abstraction.

As a second example suppose we wish to compute the first element of the reversal of a list. The specification is

$$hdrev(\textbf{nil}) \leftarrow \textbf{nil}$$

$$hdrev(cons(h,r)) \leftarrow hd(rev(cons(h, r))).$$

Using the fold–unfold methodology this program is improved to:

$$hdrev(\textbf{nil}) \leftarrow \textbf{nil}$$

$$hdrev(cons(h, \textbf{nil})) \leftarrow h$$

$$hdrev(cons(h, cons(s,t))) \leftarrow hdrev(cons(s, t)).$$

Again the idea is to exploit the context in which a list is being reversed to achieve a simplification. This example is taken from Scherlis [38]. Scherlis has reformulated the fold–unfold methodology using *expression procedures*, which allow more complex expressions on the left-hand side of equations. He has proved the total correctness of his rules and has derived nontrivial algorithms with the method, including parsing and combinatorial graph algorithms [36]. The folding rule described here does not preserve the total correctness of a program.

In these last two sections the use of low-granularity transformations for program development in functional and logic programming settings have been illustrated on "toy" examples. Development of large programs by the manual application of low-granularity transformations is not practical. In the next section we describe high-granularity transformation which encapsulate significant programming knowledge. These transformations capture design and optimization techniques that commonly appear in formal program developments.

IV. Program Improvement Techniques

The optimization techniques described in this section are relevant to programming languages with composite-valued data objects and declarative constructs, features common to many very-high-level languages. Composite data objects have values which are aggregates or collections of elements, where the elements are data objects of the language. Examples are arrays, lists, sets, sequences, records, relations and maps. Operations which directly manipulate composite data objects contribute to the notational power of very-high-level languages. Operations of this type commonly found in programming languages include the following:

reduction—The cumulative application of an associative binary operation to each element of a composite object, for example summation over the elements of a list.

apply-to-all or image—Apply a function to each element of a composite object forming a new composite object, for example forming a list of the squares of the numbers on a given list.

element test—Determine if a data value is an element of a composite object.

size—Compute the number of elements in a composite object.

selection or filter—From a subcollection of a composite object consisting of all elements that satisfy some predicate.

comprehensions or set-former—Construct a composite object by enumerating through a collection of values and performing selection and image operations. For example $\{f(x), x \in S : p(x)\}$ forms the set of all values $f(x)$ for which x is in S and $p(x)$ is true.

Many of these operations are implemented by enumerating the elements of the composite data object. Optimizing the construction of the loops which implement these operations is discussed in Section IV-B. Related to this optimization is finite differencing, which is discussed in Section IV-A. We refer to sets, sequences, records, relations, and maps, as *set-theoretic* datatypes and regard lists and arrays as datatypes that provide implementations for these types. Section IV-C discusses the representation of set-theoretic datatypes. *Maps* are data objects that associate an element in a domain set an element in a range set. Maps which are implemented procedurally as function procedures are called *functions*. Maps (necessarily finite) represented by a data structure, such as a set of domain-range pairs, are called a *stored maps*. Choosing between a stored or computed representation of a map is an implementation decision discussed in Section IV-E.

The declarative constructs of a programming language provide a mechanism to state assertions. As remarked in Section II only weak methods are known that make computational use of assertions of a general form, for example those of the full first-order predicate calculus. Stronger methods can be applied when the form of the assertion is restricted. Restriction to Horn formulas makes deduction a viable computational mechanism. Other restrictions on the form of a first-order formula can lead to efficient computational use. The restriction of formulas to conditional equations, i.e., universally quantified statements of the form $p \rightarrow (t_1 = t_2)$ has been explored [24] in connection

with algebraically defined datatypes. Terms rewriting systems may provide a computational interpretation for sets of conditional equations.

A very restricted form of assertion is a *definitional assertion*, an assertion of the form $x = e(y_1, \cdots, y_n)$. It is interpreted as defining x as an object whose value is given by the expression e. e depends on free variables $y_1, \cdots, y_n$. Queries and view definitions in relational databases are assertions of this form. The equations of a functional program are definitional assertions defining maps. Section IV-E discusses the procedural interpretations of definitional assertions.

A. Finite Differencing

Finite differencing [32], [43] is an important optimization for languages which manipulate composite objects. The optimization seeks to replace costly recomputations of such objects with cheaper, incremental updates. The method will be described in the context of maintaining definitional assertions in an imperative setting, but the method is fundamental to all styles of programming. In Bird [8] there is an abstract description of the technique in purely applicative terms.

Suppose that the definitional assertion $x = e(y_1, \cdots, y_n)$ is to be maintained in a body of code. It is useful to think of this assertion as a loop invariant. Now within the code block, which may be a loop, a free variable y_i of the expression e may be modified. Thus to maintain the assertion it is necessary to recompute x. Recomputation may be very expensive, but if the change to y_i is "small" then a corresponding "small" modification may be made to x that is cheaper than recomputation. Finite differencing is the optimization method which replaces these recomputations by incremental updates, often realizing asymptotic improvements in efficiency.

As an example suppose X is defined by the definitional assertion $X = \{z \in Y : z > a\}$. That is, X is defined by an expression, with two free variables, Y and a, and denotes the set-former that constructs the set of elements of Y that are greater than a. Suppose Y undergoes modification, for example, an element b is added to Y. To maintain the constraint the value of X must be modified to reflect the change in Y. Naively the set is recomputed according to its definition. But this "small" change in Y causes only a "small" change in X. Clearly the value of X may updated by the statement ***if*** $b > a$ ***then*** $X := X \cup \{b\}$. Finite differencing detects this improvement and provides the *update code*. The update code can be derived algebraically from the definitional assertion and the modification to the free variable. Substituting the new value of Y for the old in the definition of X yields

$$\{z \in (Y \cup \{b\}) : \ z > a\}.$$

Distributing the set-former operation over union gives

$$\{z \in Y : z > a\} \cup \{z \in \{b\} : z > a\}.$$

Folding X for its definition and simplifying the second set-former gives

$$X \cup \textbf{\textit{if}}\ b > a\ \textbf{\textit{then}}\ \{b\}\ \textbf{\textit{else}}\ \{\ \}.$$

Distributing the union over the conditional gives

$$\textbf{\textit{if}}\ b > a\ \textbf{\textit{then}}\ X \cup \{b\}\ \textbf{\textit{else}}\ X.$$

This is the new value that is assigned to X. Finally the assignment operation ($:=$) may be distributed over the conditional which yields the update code ***if*** $b > a$ ***then*** $X := x \cup [b]$. This program improvement follows the fold–unfold methodology of derivation illustrated in Section III. This improvement could be derived each time it is needed. However for frequently occurring constructs like the set-former above it is best to store a finite differencing rule rather than rederiving the rule as needed. Once a basic set of finite differencing rules have been developed, they can be combined to finite difference more complex expressions, because the finite differencing rules have algebraic properties which reflect the algebraic properties of the underlying operators.

A finite differencing step may be at heart of a sophisticated algorithm. Consider the algorithm to topologically sort a directed, acyclic graph. A *topological sort* is a total order of the vertices of the graph consistent with the partial order induced by the edges. The algorithm works by choosing as the first vertex in the ordering a vertex v which has no predecessors, i.e., for which there is no w such that (w, v) is an edge in the graph. Such a vertex is called a *minimal vertex*. It then deletes v along with any edges incident to v from the graph. The algorithm recurses until all vertices are selected in this way. For this algorithm to be efficient finding a minimal vertex must be efficient. If the set M is defined by a definitional assertion to be the set of minimal vertices, we seek to maintain this set by finite differencing when a vertex is deleted from the graph. M is defined in terms of another definitional assertion for the map *Pred*, which associates with each vertex the number of its predecessors. That is, M is the set of vertices whose *Pred* value is 0. Maintaining M is a sophisticated finite differencing problem, but one that has been solved automatically [32]. When the vertex v is deleted from the graph the value of the map *Pred* is decremented for all vertices w such that (v, w) is an edge. v is deleted from M and all vertices w for which *Pred*(w) has been decremented to zero are added to M. The modifications made to *Pred* and M when v is removed from the graph is the update code produced by finite differencing.

If $x = e(y_1, \cdots, y_n)$ is a definitional assertion and if m is a modification to one of its parameters, and if there exists update code that maintains the value of x after the modification m that is (asymptotically) cheaper than recomputation of e then e is *continuous with respect to the modification* m. Consider the example above, maintaining $X = \{z \in Y : z > a\}$. If a is decremented then it is easy to derive the update code ***if*** $a \in Y$ ***then*** $x := x \cup \{a\}$ (executed before the update to a). But if the assertion was $X = \{z \in Y : f(z) > a\}$ then, in general, no incremental update is possible (there may be many $z \in Y$ for which $f(z) = a$). This expression is said to be *discontinuous with respect to the modification*.

B. Loop Fusion

Many of the operations defined on composite objects contain implicit enumeration, which when refined to low-level operations are implemented as loops. Loop fusion is an optimization technique that seeks to merge or fuse loops generated by these operations together. This optimization will reduce loop overhead and, more significantly, avoid the need to store composite-valued intermediate results.

A program to compute the mean and variance of a list of numbers S is specified by the following definitional assertions:

$$Sum = reduce(S, \text{``+''})$$

$$Squares = apply\text{-}to\text{-}all(S, \lambda x.x^2)$$

$$SumOfSquares = reduce(Squares, \text{``+''})$$

$$N = size(S)$$

$$Mean = Sum/N$$

$$Var = (SumOfSquares - Sum^2)/N^2.$$

Naively, if each defined object is computed independently the implementation requires four loops. Three loops enumerate S to compute N, *Sum*, and *Squares*. A fourth loop enumerates *Squares* to compute *SumOfSquares*. These four loops can be combined into one by two loop-combining techniques called *vertical* and *horizontal loop fusion*. Expressions which do not depend on one another and which are computed by enumerating over the same object may be computed in the same loop by horizontal fusion. The loops to compute N, *Sum*, and *Squares* are combined by horizontal fusion. The computation of the *SumOfSquares* is fused into the same loop by vertical fusion. Suppose we are computing $G(F(S))$, where F is computed by enumerating over S yielding an intermediate list, and G is computed by enumerating over this list. Then vertical fusion computes $F(S)$ and $G(F(S))$ in a single loop. In the example, $G(F(S))$ is *SumOfSquares* and $F(S)$ is the intermediate list *Squares*. The origin of the terms vertical and horizontal fusion may be seen by examining the directed acyclic graph that represents these expressions given in Fig. 1.

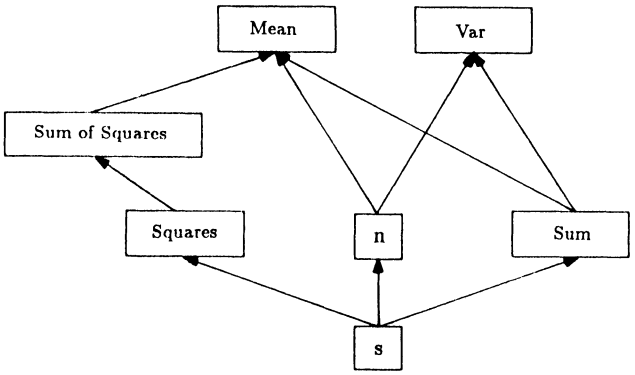


Fig. 1.

The code resulting from the fusion steps is:

```
Sum := 0;
N := 0;
SumOfSquares:=0;
Squares := nil;
for each x ∈ S do
begin
   Sum := Sum + x;
   N := N + 1;
   Squares := Squares append x^2;
   SumOfSquares := SumOfSquares + x^2
end
```

Note that *Squares* is a dead variable—it is never used anywhere in the computation—and so it can be eliminated. Elimination of composite-valued intermediate objects such as *Squares* is the chief advantage of the fusion technique.

Loop fusion is possible when two objects are computed by enumerating a composite-valued parameter. But to say that an expression can be computed by enumerating one of its parameters is equivalent to saying that that expression is continuous with respect to modification of that parameter by element addition. Thus the detection of loop fusion opportunities strongly depends on a finite differencing capability. The statements in the fused loop above are exactly the update code that maintains the definitional assertions with respect to element addition.

Observe that the more natural definition of variance, namely, $Variance = \Sigma(S_i - Mean)^2/Mean^2$ is not used precisely because this expression does not have the desired finite differencing property. The intermediate list $apply\text{-}to\text{-}all(S, \lambda\, x.\, (x - Mean)^2)$ is not continuous with respect to appending an element to S since this modification changes *Mean* and so all previous elements in the list must be recalculated. The connection between finite differencing and loop fusion is discussed in [32] and [43] and illustrated in a derivation of a garbage collection algorithm in [15].

Opportunities for the application of loop fusion to functional programs are prolific. Horizontal fusion is achieved by a tupling technique. Suppose $H(F_1(S), F_2(S))$ is to be evaluated where $F_1(S)$ and $F_2(S)$ are list-valued and computed by enumerating S. Using the tupling technique a new function $F(S)$ is defined which computes a list of pairs (tuples). The ith element of $F(S)$ is the pair consisting of the ith element of $F_1(s)$ paired with the ith element of $F_2(s)$. Computing F instead of F_1 and F_2 achieves horizontal fusion. Vertical fusion is achieved by algebraic manipulation that transforms $G(F(S))$ into $(G \circ F)(S)$ where $(G \circ F)$ is computed with only a single enumeration of S and produces no intermediate sequence.

Burstall and Darlington [11] demonstrated a loop fusion step using the fold–unfold methodology. Bellegarde [6] considers a subclass of functional operators (e.g., *apply-to-all*) for which algebraic identities can be applied to an expression to achieve a normal form in which the number of intermediate sequences needed for evaluation has been minimized. In [19] it is shown using a graph-theoretic

model that achieving best possible loop combining is NP-complete even if all relevant finite differencing opportunities are known. One source of the difficulty is that certain operations can be implemented using different iterative structures. For example, $A \cap B$ can be computed by enumerating either through A or through B (and performing a sequence of membership tests on the other), providing different fusion opportunities.

C. Data Structure Selection

In this section data structure selection techniques for the implementation of set-theoretic and algebraically specified datatypes are examined. Set-theoretic datatypes provide a means of introducing abstraction into a specification. Algorithms expressed in SETL [41], a procedural programming language with a rich collection of set-theoretic types, are generally an order-of-magnitude more concise then the equivalent algorithm in Pascal. A great deal is known about representing set-theoretic types. For example two chapters of a popular introductory text on data structures [1] is devoted to representations of sets (lists, arrays, trees, hash tables, etc.). Automatic derivation of efficient representations is a difficult task.

The selection of an efficient representation of a set-theoretic object depends primarily on the operations performed on the object, whether the elements of the object have a natural ordering associated with them, and on the relationships that exist among the objects in the program. Different representations implement operations defined on the type with varying efficiency. The representation of a set as bit-vector provides an inefficient implementation of enumeration but an efficient implementation of element tests. A set represented as a list can be enumerated quickly but element tests are slow. If the elements of a set have a natural order that can be quickly computed, as is the case for integers, then a whole range of representations that exploit the ordering (ordered lists, binary search trees, balanced tree schemes, priority queues, etc.) become available. Finally, relationships among the objects of a program, particularly inclusion information, affect data representation. If A and B are sets and $A \subseteq B$, then A can be represented as a map from B to Boolean. This map is called the *characteristic* map of the set.

The SETL language [41] provides a comprehensive collection of set-theoretic datatypes and uses a conventionally structured compiler for their translation. The compiler's default representation for sets is a linked hash table. This representation provides an asymptotically optimal (expected case) implementation of insertion, deletion, membership test, and enumeration. Linked hash tables are also the default implementation of maps. Maps are hashed on the domain value so that map application is a constant-time operation. Although hash tables provide good asymptotic performance, the overhead associated with hashing operations is considerable. Thus the compiler provides alternate representations [44] to reduce the dependence on hashing. Alternate representations are selected primarily on the basis of subset relationships among data objects of the program, and secondarily on the operations performed on the objects. The subset information is used in conjunction with the data structure selection rule stated above—"If A and B are sets and $A \subseteq B$, then A can be represented as a map from B to Boolean." Specifically the set B is called a *base*, and is represented as a hash table. Elements of the hash table are records called *element blocks*. One of the components of the element block is the value of the element. Another component of the element block represents the set A. A is represented by a Boolean-valued component of the element block, called *is-element-of-A*, which is true if the element of B represented by the block is also an element of A. Thus the characteristic map representing A is implemented as a property of the elements of the map's domain. This representation of A is called a *local basing* of A. In a modification of this representation, called an *indexed basing*, the bits held locally in the *is-element-of-A* component of the element blocks using local basing are collected into a single bit-vector. The *is-element-of-A* component of the element block contains an index into this bit-vector.

If the variable x ranges over values which are always elements of B, then x is represented as a pointer to the element block which corresponds to the value of x. If A is represented by an indexed basing, to test if x is in A, the pointer x is dereferenced to access the element block. The index into the bit-vector representing A is retrieved from the element block. The bit-vector is accessed using the index to retrieve the bit which indicates x's membership in A.

If f is a map with domain D and range R, where D is a subset of B, then f may be represented as a component, called *value-of-f*, of the element blocks of B. The type of the component is the type of elements in range set R. The value of *value-of-f* in the element block representing x is $f(x)$. This is a *local base* representation of a map. To evaluate the application $f(x)$, the pointer x is dereferenced to the element block representing the value of x, then the value of the component *value-of-f* is returned. The definition of an *indexed base* representation of a map is completely analogous to the definition of an indexed base representation of a set.

Using the local base representation, the operations map application, insertion, deletion, and membership tests are very fast. Unfortunately, other operations are slow, notably assignment of the object, passing the object as a parameter, incorporation of the object into larger composite data objects, and enumeration. To enumerate the set A, the whole base B must be enumerated. If A is a sparse subset of B then this is slow. By collecting the elements of A in a single vector, the indexed base representation supports structure sharing. This speeds up assignment, parameter passing, and the incorporation of A into larger composite data objects. Insertion, deletions, and membership tests are slowed down by the introduction of another level of indirection for element access.

For example, a natural representation of a graph uses the set of vertices as a base. The adjacency map, whose

domain is the vertex set, is represented as a component of the element block of the base, as can any subset of the vertex set, or any maps on the vertices that might be computed. The SETL compiler does not seek to find a naturally occurring data object to serve as a base, but creates a synthetic one. A base B is a compiler generated variable. If sets A and C use B as a base, then the assertions $A \subseteq B$ and $C \subseteq B$ are maintained by inserting into B any element that is inserted into A or C.

A SETL program may optionally contain declarations which specify the based representations sets and maps. An optimizing compiler that automates the selection of these representations has been developed. The analysis performed by the compiler is complex and is only briefly summarized here. First, bases are provisionally assigned to variable occurrences on which operations that benefit from a based representation are performed. Using data flow analysis, variable occurrences which should share the same base set are detected. Those bases that are shared are merged. Finally, because a variable may have different representations in different regions of code, code to convert between representations is inserted and optimized. Details may be found in [44].

In the PSI project [26] and later in the follow-up CHI project [27], [46], implementations of set-theoretic data types are derived by transformation. In the CHI system, a data type implementation results from a sequence of data structure refinements, each refinement expressing a small, independent data structuring decision. A refinement specifies the implementation of one data type in terms of another. These refinements are expressed as a set of transformation rules; one lead rule which refines the data type declaration and others which refine the operations of the data type to reflect the refinement decision. For example, one refinement rule, stated informally is "if f is a map from an integer subrange to B, then f may be represented as an array of elements of type B indexed by the subrange." Another more complex rule states "a map f from domain D to range R can be represented by a map h from domain D to range B and a map g from domain $B \times D$ to range R." This rule is used to represent the map f as a hashed map. The map h is the hash function. It will be represented procedurally (see Section IV-D). The elements of the set B are the buckets of the hash table, and the map g represents the organization of the map within each bucket. When a data type is refined all operations performed on the type must be transformed into equivalent operations on the implementing type. The map application $f(x)$ gets refined into array reference for the first rule and into the expression $g(h(x), x)$ for the second.

The CHI system contains over 120 data structure selection rules. These rules express refinements which, when composed together, are capable of generating a large space of implementations. Included in this space are the common implementations of set-theoretic types. The system can select transformations completely automatically providing a rapid prototyping capability. In this mode the system generates a default implementation based on linked lists. The PSI system, the predecessor of CHI, has an efficiency estimator [25]. The efficiency estimator is used to automatically select among the alternative data structure representations. A second generation efficiency estimator is planned for CHI.

The notion of refinement can also be used for synthesizing the implementation of algebraic datatypes. This is the approach taken by the CIP (computer-aided intuition-guided programming) project [5] at the Technical University of Munich. CIP employs a wide-spectrum language based on algebraic datatypes. A program is composed of specifications of algebraic datatypes. The specification of an algebraic datatype consists of a *signature* and a collection of *laws*. The signature consists of names for certain sets called sorts, names for the operations of the type together with names of the sorts of their operands and results. The laws are formulas of the first-order predicate calculus which axiomatize the properties of the operations. An *implementation* or *model* of an algebraic datatype provides for each sort name an actual set of objects, and for each operation a function over the sorts that satisfies the laws specified in the type definition. The notion of refinement can be used to structure the construction of an algebraic datatype implementation into smaller, more manageable tasks. A refinement describes an implementation of a type t_1 in terms of a type t_2. So if an implementation is constructed for t_2 then the refinement gives an implementation constructed for t_1 as well. An implementation for t_2 is constructed by refinement as well. This process completes when all the types are refined into types that have a machine implementation.

A refinement is specified by an *abstraction map* which maps elements of the sorts of the implementation datatype onto the elements of the sorts of original (abstract) type. If an abstraction map maps x to y then in the implementation x can represent the object y. To implement the refinement, for each operation of the abstract type a program must be constructed that implements the abstract operation in terms of operations of the implementing type. This itself is a synthesis task in a logic programming setting with the laws of the two types and the abstraction function serving as the specification. Equivalently this can be seen as deriving the transformation rules used to refine operations. From this perspective, the set-theoretic datatypes in the CHI system are algebraic datatypes for which refinements have been predefined and implemented with the system.

Another approach to implementing algebraic datatypes, suitable for rapid prototyping, directly derives machine-representable implementations from the specification of the type. The laws of the datatype must be restricted to conditional equations. These implementations are based on term models. In a term model the sorts of the datatype are sets of equivalence classes composed of the variable-free terms built from the constants and function symbols (operations) of the type. Two terms are in the same equivalence class if their equality can be proved from the laws of the type. Specialized inference procedures based on

term rewriting systems [24] can sometimes (depending on the laws) provide implementations. We omit details here.

D. Store-Versus-Compute

Suppose f is a map defined by a definitional assertion, $f = \lambda x \in D \cdot e(x, y_1, \cdots, y_n)$. The body of lambda expression, $e(x, y_1, \cdots, y_n)$ contains the bound variable x and the free variables $y_1, \cdots, y_n$. The map f may be implemented as follows.

- As a *function procedure:* Map application is refined into procedure invocation. This is the *computed* or *procedural* representation of a map.
- As a *stored map:* The map is represented as a data structure consisting of a set of domain–range pairs. Application is indexed retrieval from the data structure.
- As a *memoed map:* This is a mixed strategy which dynamically maintains a data structure of domain–range pairs for those domain points for which the value of the map has been computed. Application of the map is refined into code which first checks if the data structure contains the desired value. If it does not then the value is computed using the definition of the map, and the result is stored in the data structure and returned as the value of the application.

In this section we discuss the factors considered in choosing one of these representations. The dominant concern is the avoidance of redundant and needless computation. Computing the map introduces the possibility of *redundant computation*—recomputing the value of the map repeatedly for the same domain point. Storing the map introduces the possibility of *needless computation*—computing the map on domain points never actually required by the computation. Memoing [31] ensures that no redundant or useless computation is ever performed but requires greater overhead than the other representations.

A secondary factor affecting store-versus-compute decisions is the space–time tradeoff the choice presents. Computing a map may introduce redundant computation but this may be regarded as a lesser evil than the utilization of memory required by a stored representation.

Needless computation may be avoided by domain restriction. For the purpose of modularity and clarity the domain D of the map may be defined to be a much larger set than actually required. Restricting D to a subset D' is *domain restriction*. Once the domain is restricted storing the map may become the more attractive alternative. A common case is the following. A map is defined with domain the natural numbers. In fact the map is computed for only an initial segment $\{1 \,..\, n\}$ of the integers where n depends on the input to the program. An array whose upper extent is determined at runtime may be used to represent the map. Another common case arises in functional programs where the body of the function definition cannot contain free variables. Thus, many of the parameters passed to the function are constants in the sense that, for every function application they are passed the same value. This may be detected and the domain restricted accordingly.

Finally, it may be desirable to store a map because in doing so it may be no more expensive (asymptotically) to compute a map on all elements in its domain than it is to compute it on a single domain point. Suppose $f = \lambda x \in D, e(x, S)$. Furthermore, suppose $e(x, S)$ is computed by enumeration through S and that f is not recursive. Then f, in some cases, may be computed on all of its domain with a single efficient enumeration through S, instead of an enumeration for each element of its domain. We illustrate this by extending the example of computing elementary statistics in Section IV-B. Assume elements of the list S are restricted to be integers in the range $\{0 \,..\, 100\}$. In addition to computing the mean and variance of S, we compute $freq = \lambda x \in \{0 \,..\, 100\}.\ size([s \in S : s = x])$. $[s \in S : s = x]$ denotes the sublist of S consisting of all elements equal to x; thus, $freq(x)$ is the number of occurrences of x in S. Computing $freq(x)$ for a single point x requires enumerating through S. The following code computes *freq* on all domain points with a single iteration through S:

```
for x∈{0..100} do freq(x) := 0;
for s∈S do freq(s) := freq(s) + 1
```

The first line initializing *freq* is obtained by evaluating *freq* with S equal to ***nil***. $freq(s) := freq(s) + 1$ is the update code to maintain *freq* when s is appended to S.

As a second example suppose f is a map from domain X range Y. Then $f^{-1} = \lambda y \in Y.\ \{x \in X : f(x) = y\}$. f^{-1} can be computed by

```
for y∈Y do f⁻¹(y)=φ;
for x∈X do
   begin
      y:=f(x);
      f⁻¹(y):=f⁻¹(y) ∪{x}
   end.
```

The asymptotic savings that result from this optimization make storing a map more efficient than computing it. Detection of this optimization is a finite differencing step.

To summarize, store-versus-compute decisions are made to avoid redundant and useless computation. When conserving storage is a concern, a map may be computed even if redundant computation results. Useless computation can be avoided by restricting the domain of the map by considering the context of use of the map. Finally, a map may be stored because a "batch" computation of all of its domain elements simultaneously may be extremely efficient.

The above discussion assumed that f was not defined recursively. We consider that case here. Computing or memoing a recursive map raises no special difficulties but storing a map does. Every terminating recursive map must be based on a well-founded ordering of its domain satisfying the property that recursive calls are made to domain points which are smaller with respect to this relation. The well-foundedness of the relation guarantees termination of each map application. To store a recursive map in a data structure the domain of the map must be enumerated

according to a topological sort of the well-founded ordering. This ensures that all recursive map applications become lookups into that portion of the data structure that has already been computed.

Most of the "calls" or applications of a recursive map are initiated from within the map body itself—they are what are commonly referred to as "recursive calls." Thus the pattern of calls made to the map can be analyzed just by examining the map itself. Storing a recursive map, sometimes called *tabulation*, can be very efficient since many recursive definitions specify extremely redundant computation (e.g., the Fibonacci function). The basic structure of dynamic programming algorithms is the result of tabulating a recursive map to avoid redundant computation. Memoing a recursive map also avoids redundant computation. Unlike tabulation, memoing does not require explicit knowledge of the well-founded ordering which guarantees termination of the recursion. In [18] a graph algorithm based on depth-first search is derived as a memoed implementation of a recursive map. On the other hand, explicit knowledge of the well-founded ordering can be exploited to make tabulation space-efficient. In [7] and [12] are techniques which detect when tabulated values will no longer be referenced. The storage they occupy may be reused and the size of the data structure reduced.

E. Procedural Interpretation of Logic Assertions

As programs develop, moving from a high to a low level of description, logical assertions receive a procedural interpretation. Prolog and functional programs can be characterized as being at the low logic-level. Such programs consist of assertions, but the assertions are in a restricted form for which there exist efficient procedural interpretations. Prolog assertions are Horn formulas and functional programs are composed of definitional assertions. The procedural interpretation of these programs is provided during execution by the interpreter. These interpreters embody a single method of procedural interpretation used uniformly for all programs. In this section transformations expressing different procedural interpretations are described.

The procedural interpretation given to a program determines its *computational structure*. By this we mean the following. A program generates its output by computing a sequence of intermediate results. A low-level logic program is largely descriptive of what intermediate results are to be computed but not their order, beyond that required by the data dependencies. Determination of the order in which these intermediate values are computed is specified by providing a computational structure. For example, in a functional program the intermediate results are the actual parameters which get evaluated during execution. Inside-out and lazy evaluation are different procedural interpretations that may be used by an interpreter, each yielding a different computational structure.

The computational structure of a program determines its requirements for storing intermediate values. The need to store intermediate values within a computation arises when the sources(s) of data are not local to the points at which data are used. Green and Barstow [21] use a paradigm of *producers* and *consumers* of data. One objective in choosing a computational structure is to provide locality between producers and consumers of data to avoid storing intermediate results.

Achieving locality between the definition and uses of a composite object is a necessary but not sufficient condition to avoid storing the object. It also depends on how the object is created and how it is used. A composite object may be created by a computation in which its elements are generated one at a time and collected to form the object. Or it may be created by an incremental construction of each of its elements, as exemplified by the construction of the stored map *freq* in the previous section. The value of any of its elements is not determined until the construction of the object is complete. An object created in this way must be stored.

The use of an object may be *sequential* or *random*. A *sequential* use of an objects enumerates the elements of the object in an arbitrary order. A *random* use of an object accesses elements of the object in an unpredictable order. The need to store the object is avoided only when the object is defined by element generation and the uses of the object are sequential. Then by providing locality between definition and uses, loop fusion will remove the intermediate data object.

Maintaining objects in sorted order is a fundamental technique which allows certain operations which might have required random access to be implemented by sequential access. For example, the intersection of two lists can be computed by sequential enumeration if the lists are sorted. A file of purchase orders may be sorted by customer so that the customer file may be updated by sequentially enumerating both files.

We consider two forms of assertions, definitional assertions and Horn formulas. Suppose a data object x is defined by the definitional assertion $x = f(y_1, \cdots, y_n)$. At the logic level this is an assertion defining the relationship between x and y's. At the imperative level this is an invariant that must be maintained. Specifically, on every control path in the program between a redefinition of a y and a use of x there must be code that ensures the invariant. The invariant may be maintained by inserting the assignment $x := f(y_1, \cdots, y_n)$ after each modification of a parameter y_i. This is called *active maintenance*. A second method of maintaining the invariant is *delayed maintenance*. Delayed maintenance unfolds the definition of an object at each of its uses. Thus the invariant $x = f(y_1, \cdots, y_n)$ is maintained by substituting the expression $f(y_1, \cdots, y_n)$ for x at each use of x. Active maintenance is used when the potentially costly assignment inserted at each modification to the parameter y_i can be replaced via a finite differencing step by a cheaper incremental update.

An advantage of delayed maintenance is that it places the definition of the object into the context of its use, pro-

viding opportunities for optimization. For example, suppose x is defined by the assertion $x = \{ y \in Y : p(s)\}$ and the use of x is a membership test $z \in x$. Unfolding and algebraic simplification yields $z \in Y \,\&\, P(z)$ which avoids the construction of x. x may have been defined as an infinite object. Delayed evaluation avoids its construction.

We next turn to the procedural interpretation of Horn formulas. Recall a Horn formula is of the form $P_1 \,\&\, P_2 \,\&\, \cdots \,\&\, P_n \rightarrow Q$. Assertions of this form can be maintained by backward or forward inference. Backward inference is "demand-driven." When the assertion Q needs to be established, an attempt is made to establish $P_1 \,\&\, P_2 \,\&\, \cdots \,\&\, P_n$. Forward inference is "data-driven." When $P_1 \,\&\, P_2 \,\&\, \cdots \,\&\, P_n$ is established, an inference establishing Q is made. Prolog uses backward inference. If a Prolog program contains two assertions, $P_1 \,\&\, P_2 \,\&\, \cdots \,\&\, P_n \rightarrow Q$ and $R_1 \,\&\, R_2 \,\&\, \cdots \,\&\, R_n \rightarrow Q$ and the system seeks to prove (an instance of) Q it will attempt to do so applying backward inference to the two assertions always in order in which they appear in the program. To prove the conjunction $P_1 \,\&\, P_2 \,\&\, \cdots \,\&\, P_n$ it will always attempt to prove each conjunct in the order in which they appear. This single procedural interpretation is given to every Prolog program.

The NAIL! (Not Another Implementation of Logic!) project [47] at Stanford University seeks to optimize the implementation Prolog programs by providing alternate procedural representations, expressed in *capture rules*. Some of the defined capture rules select forward or backward inference, optimize the implementation of linear recursions, and provide strategies for proving conjunctions.

Westfold [50] has built a *logic assertion compiler* as part of the CHI system which compiles assertions into procedures. The program is annotated with *directives* to the compiler specifying how the assertions are used. A directive may specify compile-time or runtime use of an assertion. An assertion used at compile time conceptually becomes part of the compiler itself. It is used within the compiler in a manner specified by (other) directives associated with the assertion. The assertion is exploited to improve the generated code in a way analogous to the way an optimizing compiler exploits an algebraic law to reorder the computation of an arithmetic expression. This interesting capability can be realized because the logic assertion compiler compiles a program by applying transformation rules. The logic assertion and its directive is compiled into a transformation rule which is appended to the rule base of the compiler. An assertion used at runtime may be used for forward or backward inference as specified by a directive. A directive may also specify that results of an inference be cached (memoed) so it is never rederived.

F. Algorithm Design

In this final section on techniques we describe some methods of algorithm design. There is no sharp distinction between algorithm design and optimization but a rough rule of thumb is that an algorithm design step results in an asymptotic improvement of efficiency. Specifications, given at the highest level of abstraction, are generally in the form *find* x *satisfying* $P(x, y_1, \cdots, y_n)$ where $y_1, \cdots, y_n$ are the input variables. This is just a rewording of the specification given in logic as $\forall y_1 \cdots \forall y_n \exists x (P(x, y_1, \cdots, y_n))$. Two fundamental strategies for satisfying such a specification are *search* and *solution construction*. We consider solution construction first.

Most algorithms work by solution construction, that is, solutions are constructed directly rather than searched for. This is achieved by *problem decomposition*—breaking a problem into subproblems, and constructing a solution to the problem by composing the solutions to the subproblems together. These subproblems are often of the same form as the original, leading to an efficient recursive solution. This strategy is effective on a wide class of problems. A notable exception are the NP-complete problems; recursive decompositions, although possible, do not lead to asymptotic improvements over search of an exponentially large space. The most basic form of recursive problem decomposition is the divide-and-conquer scheme. Smith [45] has been able to automatically derive divide-and-conquer algorithms for problems of nontrivial complexity. He used divide-and-conquer to synthesize a collection of sorting algorithms. He starts with the logic specification *find* x *satisfying* $(perm(x, s) \,\&\, ordered(x))$ and uses deductive techniques to generate a functional program. The deductive steps are guided by a control strategy which seeks to instantiate a divide-and-conquer schema. The schema is

$$F(x) = \textbf{if } Primitive(x) \textbf{ then } DirectlySolve(x) \textbf{ else } Compose \circ (G \times F) \circ Decompose(x)$$

If x is not primitive, then x is decomposed (by *Decompose*) into a pair of objects. The left and right components of the pair are operated on by G and F, respectively, yielding a new pair which is operated on by *Compose*. In its most classical form G is instantiated to F. For example, mergesort is seen as an instance of this scheme with $Primitive(x)$ testing if x is the empty sequence or a sequence with one element. If x is primitive it is already sorted so *DirectlySolve* is the identity function. Otherwise *Decompose* yields a pair whose first component is the first half of the sequence and whose second component is the second half of the sequence. G is instantiated to F so both subsequences are sorted recursively. Finally *Compose* is instantiated to the merge operation yielding a sorted sequence.

In Smith's scheme synthesis is performed by proposing either a decomposition or a composition function and deriving the other from the definition of the schema, the assertions in the specification, and datatype axioms. In the most elementary application of the schema the proposal for the composition or decomposition function is datatype driven. The decomposition (composition) function is instantiated to an elementary operation of the input (output) datatype. For example to derive *quicksort* concatenation

	Singleton split	Equal size split
Work done by compose	Insertion	Mergesort
Work done by decompose	Selection	Quicksort

Fig. 2.

of sequences is proposed as the composition function. The specification of *Decompose* is automatically derived. The derived specification for *Decompose* asserts that the sequence must be partitioned into two subsequences such that all the elements of the first subsequence are less than or equal to all the elements of the second. Thus when the two subsequences are sorted concatenation results in a sorted sequence. An algorithm for *Decompose* meeting the derived specification must be synthesized. Divide-and-conquer applies to this specification as well and is used to synthesize the algorithm that partitions the original sequence into subsequences.

Recall in the schema the function G is unspecified. Instantiating it to F yields divide-and-conquer algorithms that divide the original problem into two subinstances of the original specification. Another class of algorithms results from instantiating G to the identity function. In this case the schema yields a linear recursive algorithm. Typically the proposed composition or decomposition function divides a composite object into a single element and the composite object with that element removed. For the sorting problem a sequence may be decomposed into its first element and its tail. The corresponding composition function that is synthesized inserts the first element into the tail of the sequence which has been sorted recursively. This is *insertionsort*. Alternatively, the composition function may be specified as inserting an element to the front of the sequence and the decompose function synthesized. The result is *selectionsort*. The total correctness of these algorithms is guaranteed by providing a well-founded relation which can be used to prove the recursion terminates.

The relationship of the four sorting algorithms to divide-and-conquer was first reported in Green and Barstow [21]. These results are summarized in the Fig. 2. These sorting algorithms, except for mergesort, may sort in-place. Green and Barstow discuss principles to derive low-level implementations that sort in-place. Smith generates functional programs. Reasoning about storage utilization based on an array implementation of sequences is not considered at that level.

Heapsort may been seen as a refinement of *selectionsort* where the sequence is stored in a heap to improve the efficiency of the selection of the minimal element. Derivation of *radixsort* depends on viewing the keys themselves as composite objects, tuples of fixed length in a product space over finite sets. This algorithm is derived by problem reduction via decomposition of the keys.

There is a strong relationship between finite differencing and the divide-and-conquer schema resulting in linear recursions. Suppose to compute $F(x)$, where x is a list, divide-and-conquer is applied with a decomposition function which divides the list into its head and tail, and G is instantiated to the identity. Now suppose $y = F(x)$ is a definitional assertion. The update code which maintains this assertion when an element is prepended to x must meet the same specification as the composition function used in the divide-and-conquer solution.

We now turn to the construction of algorithms utilizing search. Starting with the specification *find x satisfying* $P(x, y_1, \cdots, y_n)$ a naive search strategy will first construct a program that computes the predicate P. It then seeks to find a set S such that if there is an x satisfying P then x is in S. The solution is then found by enumerating through S and testing P. Such an algorithm is called a *generate-and-test* algorithm. Generate-and-test algorithms may be improved by *filter promotion*. The predicate P may be a conjunction of conditions. Each of these conditions act as a "filter" trapping generated values that do not satisfy the specification. Filter promotion incorporates the filter into the generator so that values which will not satisfy a conjunct of P are never generated.

Simple search based on generate-and-test does not exploit the internal structure of a composite-valued output object but more sophisticated search algorithms, such as backtracking, do. A backtracking algorithm constructs a partial output and when it recognizes that the partial output cannot be extended to a full solution it backtracks, thus eliminating a fruitless search over a large chunk of the search space.

Sharir [43] presents techniques that can make substantial improvements to algorithms based on search. He starts with the specification *find* $X \subseteq R$ *satisfying* $P(X)$, where P states a closure condition[1] on X. This is first refined into an algorithm which uses backtrack search. Such an algorithm starts with X empty, and adds arbitrarily chosen elements from R into X until $P(X)$ is true or backtracking becomes necessary. He states general conditions on P which can be used to improve the algorithm so that backtracking is unnecessary. The improved algorithm only adds elements to X that insure that X may be extended to a full solution. To satisfy the closure condition insertion of an element into X may require other elements be inserted into X as well. In a further refinement, those elements which may force further insertions into X are stored in a set called a *workset*. The workset may be maintained by finite differencing.

As an example of a derivation of an algorithm that follows this development consider the problem of computing the set X of all vertices reachable from a given vertex in

[1] X must be the least-fixedpoint solution of an equation of the form $f(X) = X$.

a graph. Backtracking is avoided by only inserting into X vertices reachable from those already in X. The workset contains all vertices placed in X which may have an edge to a vertex not in X. The workset is maintained by finite differencing. The algorithm removes a vertex from the workset and adds all vertices adjacent to it but not yet in X to the workset and to X. Representing the workset as either a queue or a stack leads to the breadth-first and depth-first search algorithms respectively.

We close this section with a brief discussion of nondeterminism and generalization. High-level programs often contain nondeterministic operations, for example, choosing an arbitrary element from a set or enumerating a set in an unspecified order. When the operations get refined into deterministic ones, the deterministic implementation may be chosen in a way that simplifies and improves the efficiency of the program. Enumeration orders for sets may be left unspecified until a representation for the set is chosen. The enumeration used by implementation is the one most convenient for the chosen representation. Less trivially, arbitrary selection of an element from a set denoted "*select x such that* $x \in S$," may be refined to "*select x such that* $x \in S$ & $Q(x)$" provided $\exists x(x \in S \ \& \ Q(x))$. Then the program may be refined exploiting the fact that $Q(x)$ is true of the element selected. In Sharir's development above, backtacking is removed from the algorithm because the set X is augmented with an element satisfying Q and not with an arbitrarily chosen element of R. The predicate Q is defined so that it can be proved that backtracking will not occur.

Algorithm construction based on recursive problem decomposition is perhaps the most commonly applied algorithm design technique. In mathematics it is often the case that a proof by induction is easier to construct when the theorem to be proved is generalized. The generalized theorem provides a more powerful induction hypothesis. The same observation applies to program construction. It is often easier to construct a recursive problem decomposition for a generalization of a problem than it is for the problem itself. Finding the "right" generalization that leads to an efficient algorithm is often a creative step. Schmitz [40] proposes finding generalizations by replacing constants appearing in specifications by variables. He derives different transitive closure algorithms as a consequence of generalizing on different constants appearing in the specification.

Other algorithm design methods such as branch-and-bound, greedy methods, convergence-of-approximations, etc. await even the most preliminary formalization. For a discussion of dynamic programming, a widely used algorithm design technique, see [17]. There is a substantial literature of derivation of nontrivial algorithms, many for advanced combinatorial problems. Representative of this work are [34], [10], [36], [33], and [14].

VI. Conclusions

We have surveyed some of the more important algorithm design and optimization techniques. We have not been comprehensive, notably we have not discussed recursion removal [2] nor storage optimization techniques. It is our wish to see this knowledge embedded in a knowledge-based software assistant that makes dramatic improvements in the productivity of software development. Toward this end we list the following open problems and research topics.

- Further development and understanding of optimization techniques such as those described here. The incorporation of these techniques in compilers and transformational systems.
- Structuring the process of program development. A knowledge-based assistant should provide a medium for development that supports the developer's cognitive model of the development process. Manual applications of low-granularity transformations obscure significant development decisions. Two solutions, not inconsistent with one another, are the use of high-granularity transformations and the structuring of program development by introducing a sequence of intermediate language levels. Development occurs in stages, each stage transforming a program down to the next language level. Boyle used this strategy in the TAMPR system which automatically compiles Lisp into Fortran by transformational techniques.
- Reusability by replay. In this model, maintenance is performed on the specification. The implementation of the modified specification is achieved by replaying the previous development and modifying is only where the change in specification requires. Replay is not yet a well-understood procedure.
- Parallel and distributed computation. The concerns and techniques for developing programs for concurrent architectures are substantially different than those for serial machines. Knowledge-based support for developing software for these architectures is highly desirable.
- Integration of different programming paradigms. Object-oriented programming, logic programming, functional programming, and other programming styles have varying appeal over a wide class of problem domains. Can a wide-spectrum language consistently support all of these styles?
- Use of domain-specific knowledge. A programmer uses knowledge about programming and knowledge about the problem domain to construct a program. How should domain-specific knowledge be embedded in the system? How should the knowledge be modularized? How can it be reused over a range of related application programs?

There is strong economic motivation for developing solutions to these intellectually stimulating problems. Rapid and fruitful developments can be expected.

Acknowledgment

I would like to thank W. Polak, A. K. Pröfrock, and D. Smith for reading an earlier draft of this paper. I would like to acknowledge the support and influence of all the members of Kestrel Institute. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing official policy, either

expressed or implied, of DARPA, ONR, or the U.S. Government.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

[2] J. Arsac and Y. Kodratoff, "Some techniques for recursion removal from recursive functions," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, pp. 295-322, Apr. 1982.

[3] R. Balzer, "Transformational implementation: An example," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 3-14, Jan. 1981.

[4] R. Balzer, T. Cheatham, and C. Green, "Software technology in the 1990's: Using a new paradigm," *Computer*, 1983.

[5] F. Bauer *et al.*, "Programming in a wide spectrum language: a collection of examples," *Sci. Comput. Program.*, vol. 1, pp. 73-114, 1981.

[6] F. Bellegarde, "Rewriting systems on FP expressions that reduce the number of sequences they yield," in *Proc. ACM Conf. Lisp and Functional Programming*, 1984.

[7] R. Bird, "Tabulation techniques for recursive programs," *Comput. Surveys*, vol. 12, no. 4, pp. 403-417, 1980.

[8] ——, "The promotion and accumulation strategies in transformational programming," *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 4, pp. 487-504, Oct. 1984.

[9] J. Boyle and M. Muralidharan, "Program reusability through program transformation," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 574-588, Sept. 1984.

[10] M. Broy and P. Pepper, "Combining algebraic and algorithmic reasoning: An approach to the Shorr-Waite algorithm," *ACM Trans. Program. Language Syst.*, vol. 4, no. 3, pp. 362-381, July 1982.

[11] R. Burstall and J. Darlington, "A transformational system for developing recursive programs," *J. ACM*, vol. 24, no. 1, Jan. 1977.

[12] N. Cohen, "Eliminating redundant recursive calls," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 265-299, July 1983.

[13] J. Darlington, "Functional programming," in *Distributed Computing* (APIC Studies in Data Processing, No. 20), F. Chambers, D. Duce, and G. Jones, Eds. New York: Academic, 1984.

[14] E. Deak, "A transformational derivation of a parsing algorithm in a high-level language," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 23-31, Jan. 1981.

[15] R. Dewar, M. Sharir, and E. Weixelbaum, "Transformational derivation of a garbage collection algorithm," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 650-667, Oct. 1982.

[16] L. Eriksson, "Synthesis of a unification algorithm in a logic programming calculus," *J. Logic Program.*, vol. 1, no. 3, pp. 3-18, 1984.

[17] A. Goldberg, "Derivation of dynamic programming algorithms," Kestrel Inst., Palo Alto, CA, Tech. Rep., 1983.

[18] A. Goldberg and R. Jüllig, "A strategy for semi-automatic program development," in *Proc. 19th Annu. Asilomar Conf. Circuits, Syst., and Comput.*, IEEE, 1985.

[19] A. Goldberg and R. Paige, "Stream Processing," in *Proc. ACM Conf. Lisp and Functional Program.*, 1984.

[20] C. Green, "An application of theorem proving to problem solving," in *Proc. Int. Joint Conf. Artificial Intell.*, May 1969, pp. 219-239.

[21] C. Green and D. Barstow, "On program synthesis knowledge," *Artificial Intell.*, vol. 10, pp. 241-279, 1978.

[22] A. Hansson, "A formal development of programs," Dep. Comput. Sci., Univ. Stockholm, Sweden, 1980.

[23] S. Haradhvala, B. Knobe, and N. Rubin, "Expert systems for high quality code generation," in *Proc. 1st Conf. Artificial Intell. Applicat.*, IEEE Comput. Soc., Dec. 1984, p. 312.

[24] G. Huet and D. Oppen, "Equations and rewrite rules: A survey," SRI, Menlo Park, CA, Tech. Rep. CSL-111, 1980.

[25] E. Kant, "Efficiency considerations in program synthesis: A knowledge-based approach," Ph.D. dissertation, Stanford Univ., Stanford, CA, publ. by UMI, 1979.

[26] E. Kant and D. Barstow, "The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 458-471, Sept. 1981.

[27] G. Kotik, "Knowledge-based compilation of high-level datatypes," Kestrel Inst., Palo Alto, CA, Tech. Rep., 1983.

[28] R. Kowalski, *Logic for Problem Solving*. Amsterdam, The Netherlands: North-Holland, 1979.

[29] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 90-121, Jan. 1980.

[30] ——, "Deductive synthesis of the unification algorithm," *Sci. Comput. Program.*, vol. 1, pp. 5-48, 1981.

[31] J. Mostow and D. Cohen, "Automating program speedup by deciding what to cache," in *Proc. Int. Joint Conf. Al*, pp. 165-172, 1985.

[32] R. Paige and S. Koenig, "Formal differentiation of set-theoretic expressions," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 402-454, 1982.

[33] H. Partsch, "An exercise in the transformational derivation of an efficient program by joint development of control and data structure," *Sci. Comput. Program.*, vol. 3, pp. 1-35, 1983.

[34] ——, "Structuring transformational developments: A case study based on Early's recognizer," *Sci. Comput. Program.*, vol. 4, pp. 17-44, 1984.

[35] H. Partsch and R. Steinbrüggen, "Program transformation systems," *Comput. Surveys*, vol. 15, no. 3, pp. 199-236, Sept. 1983.

[36] J. Reif and W. Scherlis, "Deriving efficient graph algorithms," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1982.

[37] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, July 1983.

[38] W. Scherlis, "Program improvement by internal specialization," in *Proc. ACM Conf. Principles of Program. Lang.*, 1981, pp. 41-49.

[39] W. Scherlis and D. Scott, "First steps toward inferential programming," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1983.

[40] L. Schmitz, "An exercise in program synthesis: algorithms for computing the transitive closure of a relation," *Sci. Comput. Program.*, vol. 1, pp. 235-254, 1982.

[41] J. Schwartz, "On programming: An interim report on the SETL project," Courant Inst. Mathematical Sciences, New York Univ., 1975.

[42] ——, "Internal, external and pragmatic influences: Technical perspective in the development of programming languages," Courant Inst. Mathematical Sciences, New York Univ., Tech. Rep., 1980.

[43] M. Sharir, "Some observations concerning formal differentiation of set-theoretic expressions," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, pp. 196-225, 1982.

[44] E. Shonberg, J. Schwartz, and M. Sharir, "An automatic technique for selection of data representations in SETL programs," *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 2, pp. 126-143, Apr. 1981.

[45] D. Smith, "The design of divide-and-conquer algorithms," *Sci. Comput. Program.*, vol. 5, pp. 37-58, 1985.

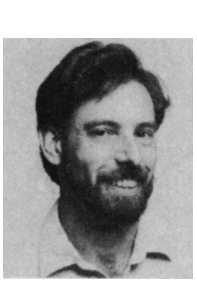
[46] D. Smith, G. B. Kotik, and S. J. Westfold, "Research on knowledge-based software environments at Kestrel Institute," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1278-1295, Nov. 1985.

[47] J. D. Ullman, "Implementation of a logical query language for databases," *ACM Trans. Database Syst.*, vol. 10, no. 3, pp. 289-321, 1985.

[48] R. Waldinger and R. Lee, "PROW: A step toward automatic program writing," in *Proc. Int. Joint Conf. Artificial Intell.*, pp. 141-152, May 1969.

[49] R. C. Waters, "The programmer's apprentice: A session with KBEmacs," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1321-1336, Nov. 1985.

[50] S. Westfold, "Very high-level programming of knowledge-based schemes," in *Proc. AAAI Conf. Artificial Intell.*, 1984.



Allen T. Goldberg received the B.S. degree in mathematics from the State University of New York at Stony Brook in 1972 and the M.S. and Ph.D. degrees in computer science from New York University, New York, NY, in 1974 and 1979, respectively.

In 1979 he joined the Computer and Information Sciences faculty at the University of California at Santa Cruz. He has been a consultant at Kestrel Institute since 1981. His interests are knowledge-based programming, programming language theory, and computational complexity.