AUTOMATIC PROGRAMMING: PAST, PRESENT AND FUTURE

J. Darlington

Rapporteur: Mr. S. Jones

Abstract

The aim of automatic programming is to produce systems that can take over many of the tasks presently performed by programmers. Thus work in this area draws on and has influence on work in Artificial Intelligence and more traditional Computer Science.

Early efforts at producing programs automatically grew out of work on automatic theorem proving and relied on the use of first order predicate logic to specify the input-output behaviour of the required program. Later work has realised the importance of the specification process and widened the scope of both the specification languages and deductive mechanisms used. Program transformation grew out of the recognition that programs could serve as specifications if they were written for maximum clarity instead of efficiency, and represents a more flexible approach to the automatic programming problem.

This lecture will survey the work done in these areas, describe some of the more successful approaches in more detail and try to evaluate the lessons learnt. We will discuss what difficulties need to be overcome, and make predictions of the impact this work will have on Computer Science and Artificial Intelligence.

Introduction

Writing programs is a task requiring intelligence. It is therefore a legitimate domain for artificial intelligence workers to investigate and has the advantage of being an area where they themselves are often domain experts. The ability to develop and modify programs or plans to cope with unforeseen eventualities is also an essential capability for an intelligent entity.

In these talks I will discuss some recent research directed towards developing systems to produce programs automatically. Much of traditional computer science, for example the development of high level languages and compilers, can be classified as automatic programming, but for the purposes of these talks I will take a narrow view of the term and concentrate on work that at least originated within the artificial intelligence sphere. It is my thesis that although this work has not produced automatic systems that come anywhere near the competence of the average programmer (and is not

likely to unless dramatic advances are made) it has provided revealing insights into the programming process and has developed techniques and systems that promise to dramatically increase programmer productivity and alter the skills and techniques he needs in such a way as to require a fundamental change in the way programming is taught.

Early attempts at automatic programming were based on the application of the resolution-based theorem provers developed during the late sixties. Work in this direction was generally abandoned during the 70's with the growing disillusionment with the power of the theorem provers available. Program transformation, an alternative that offers a more flexible approach and requires less theorem proving effort, grew in favour during the late 70's. Logic Programming was also developed in the 70's and is based on the realisation that first-order predicate calculus could be used directly as a programming language, given suitable theorem provers. Work in these latter two areas is still active and in my opinion represents the most promising approaches to automatic programming to date.

In these talks I will concentrate on giving a flavour of some of the work going on in these areas and try to indicate how I think this will alter the nature of programming. In doing this I will be ignoring substantial research efforts that are contributing to the automatic programming effort. In particular I will say nothing about work in the natural language specification of programs.

1. Automatic programming using resolution-based theorem provers

The development of the resolution principle in the middle sixties, (Robinson 1965), stimulated much work in the development of mechanical theorem provers and led to them being applied to the problem of synthesising programs from specifications.

The resolution principle offers a method for demonstrating the unsatisfiability of a set of sentences of first-order predicate logic, provided the sentences are arranged in a special way known as clausal form (Robinson 1965). Thus, to prove that a theorem is logically implied by a set of axioms, the negation of the theorem is added to the axioms and this set put into clausal form. If the theorem prover is able to prove the unsatisfiability of this set then the theorem logically follows from the axioms. It was soon realised that such theorem provers could be used in a constructive way, that is as well as proving sentences of the form $\exists x\ p(x)$ from a set of sentences S, they could also be modified to produce an instance of the x that exists. This was made possible by the answer extraction process (Green 1969). With proper formalisation the answer extraction process can be used for program synthesis. The first stage is to specify the required program by giving the relation between its input x and output y, say R(x,y). Using the answer extraction process, a

theorem proving system will produce the desired program if it can prove that the conjecture $\forall x\, \exists y\, R(x,y)$ logically follows from the axioms defining the interpretation of R. The program will consist of an example of the conjectured y as a composition of primitive functions.

As a very simple example, let us consider a program to find the maximum of two numbers. Thus our input-output relation R between the input variables x1 and x2 and the output variable z is

$$R(x1, x2, z) \equiv (z=x1 \vee z = x2) \wedge z \geq x1 \wedge z \geq x2$$

and the corresponding theorem is

$$\forall x1\, \forall x2\, \exists z\, [(z=x1 \vee z=x2) \wedge z \geq x1 \wedge z \geq x2]$$

Informally the proof proceeds by case analysis. Translating the theorem into disjunctive normal form (a stage on the way to clausal form) we have

$$\forall x1\, \forall x2\, \exists z\, [(z=x1 \wedge z \geq x1 \wedge z \geq x2) \vee (z=x2 \wedge z \geq x1 \wedge z \geq x2)]$$

If we assume $(u=v) \supset (u \geq v)$ as an axiom we can simplify the above to

$$\forall x1\, \forall x2\, \exists z\, [(z=x1 \wedge z \geq x2) \vee (z=x2 \wedge z \geq x1)]$$

Now if $x1 \geq x2$, letting z be x1 satisfies the first disjunct. On the other hand if $x1 < x2$ then $x2 \geq x1$, so letting z be x2 satisfies the second disjunct. The answer extraction process records which assignments have been made to the output variable and under what conditions they were made. Thus in this case the program extracted would be

$$\underline{\text{if}}\ x1 \geq x2\ \underline{\text{then}}\ z := x1\ \underline{\text{else}}\ z := x2$$

Several program writing systems based on this principle were written; see Green (1969), Waldinger (1969) and Waldinger and Lee (1969). They all managed the synthesis of some small programs but were limited by the power of the available theorem provers. They also had a fundamental limitation in that the method was incapable of producing programs with loops as this would require an application of mathematical induction, which is not expressible within first order logic. Various methods were proposed to get round this difficulty, generally by dealing with induction at the meta-level and using a theorem prover to construct the straight line portions of the program (see, for example, Manna and Waldinger 1971). For these reasons, work in this direction was generally abandoned. Recently, however, Manna and Waldinger have revived the theorem proving approach based on an elegant system that allows theorem proving to be carried out on

sentences without putting them into clausal form, and also admits induction. Manna and Waldinger (1980) describes this system and contains examples of the synthesis of recursive programs. As yet there is no implementation for this system although the authors state that their approach is machine oriented and ultimately intended for automatic synthesis systems. This system also allows program transformation (see next section) to be carried on in a theorem proving framework.

2. Program Transformation

The aim of automatic programming is to derive programs systematically, and hopefully automatically, from a specification. As we have seen earlier, initial attempts used predicate logic to describe the input-output behaviour of the required program and resolution theorem provers to derive the programs. The major problem that confronts this approach is the lack of sufficiently powerful theorem provers to derive anything but the simplest of programs, a consequence partly of the wide gap between the specification and target languages.

Program transformation is a hopefully more flexible approach to the same problem. It arises from the recognition that programs themselves can be specifications (and specifications can be programs) if they are written to be as clear and understandable as possible. This can best be achieved if considerations of efficiency are initially ignored and the programs written to describe what the desired result should be and not how it is to be achieved. Given such a specification the second stage of the transformation approach is to successively transform the program to more computationally feasible versions using methods guaranteed to preserve the intent of the program while improving its efficiency. Two immediate advantages of this method are that a wider variety of techniques and languages are available for the specification process and that the degree of difficulty of the synthesis can be varied continuously in contrast to the all or nothing behaviour of the theorem proving approach. Another important advantage of having programs as specifications is that they can be run, albeit probably very slowly, and thus checked or verified. The notion that specifications are simple to write and will be obviously correct only applies to small examples. The science of writing, understanding and manipulating specifications is going to become a vitally important activity.

In this section I will illustrate the nature and potential of the transformation approach by describing some aspects of my own work on transformation, work that started in collaboration with R.M. Burstall in the Department of Artificial Intelligence at Edinburgh University. Fuller descriptions can be found in Burstall and Darlington (1977), Darlington (1975) and Darlington (1977). One important aspect of this work is that I have concentrated on the transformation of applicative languages. This is for two interconnected reasons. Firstly, if we are going to manipulate

programs as objects it is advantageous if they possess respectable mathematical properties, referential transparency in particular. I also believe that the possession of these properties makes the applicative languages superior to the more machine-oriented imperative languages by being more expressible, more comprehensible and therefore better as specification vehicles. Backus (1979) makes this point forcibly. Almost all the important manipulations can be carried out within these languages even if the final implementation is to be in a conventional imperative language. Our work at Edinburgh led us to develop a purely applicative or non-procedural language called NPL (Burstall 1977). This language is used for the initial specification of our program and transformations map programs in NPL to programs in NPL. (NPL is a first order language. It has now been developed into a higher-order language called Hope by Burstall, MacQueen and Sannella (1980). The transformation techniques apply equally to NPL or Hope).

Programs in NPL consist of a sequence of recursion equations. Separate equations can be written for separate cases of a parameter. Thus, for example, the factorial function would be written

```
fact(0) <= 1

fact(n+1) <= (n+1)*fact(n)
```

and a function to square every number of a list of numbers, where the list is constructed using cons and nil (the empty list)

```
square(nil) <= nil

square(cons(n,l)) <= cons(n↑2,square(l))
```

Given such a simple language the transformation methodology can be correspondingly simple : it is detailed in Burstall and Darlington (1977) and has become known as the 'fold/unfold system'. Unfolding consists simply of replacing a function call by its definition, that is, using an equation in the normal left to right manner. Folding is just the opposite: using an equation in a right-to-left manner, replacing an instance of a right-hand side of an equation by the corresponding instance of the left-hand side.

For example, say we wanted a program to sum the squares of a list of numbers. Naively, we could write this as

```
1.     sumsquares(l)<= sum(square(l))

2.          sum(nil)<= 0

3.    sum(cons(n,l))<= n+sum(l)

4.       square(nil)<= nil

5. square(cons(n,l))<= cons(n↑2,square(l))
```

To produce an improved version we must consider the various cases of l. Firstly, letting l be nil, we can instantiate equation 1, getting

```
sumsquares(nil)  = sum(square(nil))
```

We can now use equation 4 to unfold square(nil), getting

```
sumsquares(nil)  = sum(nil)
```

and equation 2 to unfold sum(nil) getting

```
6. sumsquares(nil)  = 0
```

Returning to equation 1 we now instantiate l to cons(n,l) getting

```
sumsquares(cons(n,l))  = sum(square(cons(n,l)))
```

Using equation 5 to unfold square(cons(n,l)) we get

```
sumsquares(cons(n,l))  = sum(cons(n 2,square(l)))
```

and equation 3 unfolds this further to

```
sumsquares(cons(n,l))  = n 2+sum(square(l))
```

Finally, we observe that we can use equation 1 to fold the sum(square(l)) in the right-hand side of the above, getting

```
7. sumsquares(cons(n,l))  = n 2+sumsquares(l)
```

Equations 6 and 7 now provides an alternative to equation 1 as a program for sumsquares. As we have derived 6 and 7 from the original by simple equality substitutions we can be sure that our new program is as correct as the original and we can see that we have made an improvement in its efficiency. This gain in efficiency has been achieved by interweaving computations that were kept separate for reasons of clarity in the original specification. Of course, for this trivial example, any competent programmer could produce our final version straight off without any trouble, but consider the more realistic case of a multipass versus single pass compiler or a file transaction program where the separate transactions are done one after another versus a file transaction program that does everything in a single complicated pass over the file. Both these have exactly the same structure as our example.

NPL also allows the use of set and logic constructs as the right-hand side of equations. In Hope these can be defined directly as second order functions. This enables one to write specifications of programs even more succinctly, as in the definition of prime:

```
prime(n) <= forall m : 1<m, not(div(m,n))

div(m,n) <= exists n1 : 1<n1, n1*m=n
```

The fold/unfold system can be used to transform such programs to more conventional recursions and hence to efficient programs; see Darlington (1975). Note that the use of such set and logic constructs allows one to write original specifications that are unrunnable in that they involve searches through infinite sets. In this case we more often speak of program synthesis rather than program transformation although there is no sharp distinction between the two.

Manna and Waldinger (1975) describe an implemented program synthesis system that converts program specifications written in first order predicate calculus augmented with set constructs to runnable programs. This is achieved by directly manipulating the specification as in transformation and Manna and Waldinger independently developed a rule similar to folding which they called recursion introduction. This system achieved the synthesis of several small programs fully automatically.

2.2. The manipulation of abstract data types

Data abstraction is an important technique in program development. The techniques of equational specification (Guttag 1976) allow one to specify the behaviour of an abstract type independently of any implementation. The combination of an equational language and transformation system allows one to exploit this technique to the full. Thus an abstract data type can be specified equationally within NPL or Hope and programs written solely in terms of the abstract data type can be run and tested before any implementation for the abstract data type is considered. (Occasionally equational specifications for the abstract data type will be in a form that do not guarantee termination. These specifications can still be run however and information gathered about the abstract program). When the programmer is satisfied of the correctness of his abstract program he can use transformation to synthesise implementations for the abstract operations that are guaranteed correct. To do this he need only specify, via a representation function mapping concrete objects to abstract objects, how he wishes to represent his abstraction and implementations for all the abstract functions can be synthesised. Darlington (1977) contains details of how this is achieved and examples of successful semi-automatic syntheses.

2.3. Applications of Transformation

Given that we have methods available to manipulate programs there are various ways we can employ them.

(i) Derivation of programs

There is a wide spectrum of ways that transformation can be used in the production of correct and efficient programs. At the extremes are its use as a purely pen and paper discipline, and the artificial intelligence dream of a fully automatic system into which specifications are fed and out of which drop efficient programs. More realistic uses lie somewhere in between, leaving most of the intellectual initiative with the user but providing systems to relieve him of the book-keeping tedium, and providing ways to structure and record his thinking. The systems described in Darlington (1977) and Manna and Waldinger (1975) are examples of the artificial intelligence kind, while the systems described in Feather (1979) and Bauer et al (1977) are of the intermediate variety. Feather's system has achieved the transformation derivation of several substantial programs including a compiler for a simple block structured language and the text formatter from Kernighan and Plauger (1976).

(ii) Algorithm understanding

Transformation can be used to increase our understanding of the nature of algorithms or classes of algorithms. Systematic derivation of algorithms from their specifications can expose the main ideas or concepts behind them and the interrelationships between similar algorithms. Examples of this can be found in Darlington (1978), Clark and Darlington (1980), de Roever et al (1979) and Schmidt (1978).

(iii) Algorithm communication

Given that we have developed structured methods for recording transformations we can communicate algorithms in an understandable but completely formal way as specification plus transformation.

(iv) Program modification and maintenance

A terrible proportion of the overall programming effort is spent in modifying and maintaining existing software. Transformation offers a way of improving this position. Programs would be recorded as a specification plus a structured description of the transformation used to derive the program. Modifications would be restricted to the specification which hopefully would be understandable and modular enough to make this an easier and less-error prone exercise than at present. The transformation plan can then be rerun to produce a suitably modified program.

Hopefully moderate modifications to the specifications will not totally invalidate the transformation and the overall effort will be reduced and program reliability vastly increased. Darlington and Feather (1979) record some early experiments along these lines.

2.4 The future of transformation and its influence on programming teaching

Program transformation is not going to do away with the craft of programming unless startling advances are made in the area of combinatorics. However, I do believe that it offers an opportunity for dramatically improving the programmer's productivity and if adopted will bring about changes in his role and the training he receives. His job will become more that of a problem specifier and algorithm engineer. Students should be taught the techniques of writing and evaluating specifications in a variety of high level formalisms, and the principle and practice of developing efficient programs from them. Not only would this be of benefit as a way of programming, but I believe that it has the great advantage that algorithm study and computational complexity can be taught in a very concrete and practical way, and that it provides a valuable unification of the art of program development and the science of algorithm study and complexity. I feel that it is important that students are taught to regard programs as first-rate mathematical objects, about which precise statements can be made and which can be manipulated formally.

The programmer of the future is going to have a lot more automated tools at his disposal for the design and evaluation of programs. These will not remove the need for him to be intelligent and well trained, but properly used should increase his productivity and reliability. Any course in programming should involve an appreciation of the potentialities and pitfalls of such systems.

One of the safest predictions to make is that the range of applications of computers is going to increase and that a lot of potential users will be experts in the area of application rather than skilled programmers. Transformation can help to avoid some of the pitfalls that this entails. A general specification language such as NPL or predicate logic can be extended to provide specialised specification languages that would allow these domain experts to express their requirements in their own terms. Similarly transformation systems can be extended to achieve the efficient implementation of these specifications. I foresee the role of the applications programmer changing to the design of extensions to specification languages and transformation systems, and that particular application programs will be produced much more reliably at the point of use.

3. Logic Programming

The resolution based theorem provers discussed earlier were employed to derive programs from specifications written in first order predicate calculus. The programs thus derived were in a separate language. Recent developments have changed the emphasis somewhat with the growing realisation that predicate logic itself can

be used as a programming language by giving it a procedural interpretation and that theorem provers can be used as interpreters for logic programs.

In the clausal form of predicate logic, simple assertions are expressed by clauses :

Father(Zeus,Ares) ←

Mother(Hera,Ares) ←

Father(Ares,Harmonia) ←

Mother(Semele,Dionisius) ←

Father(Zeus,Dionisius) ←

etc.

Here Father (x, y) states that x is the father of y and Mother (x, y) states that x is the mother of y.

Clauses can also express general conditional propositions :

Female(x) ← Mother(x,y)

Male(x) ← Father(x,y)

Parent(x,y) ← Mother(x,y)

Parent(x,y) ← Father(x,y)

These state that :

x is female if x is mother of y,

x is male if x is father of y.

x is parent of y if x is mother of y, and

x is parent of y if x is father of y.

The arrow ← is the logical connective "if", "x" and "y" are variables representing any individuals; "Zeus", "Ares", etc. are constant symbols representing particular individuals; "Female", etc. are predicate symbols representing relations among individuals. Variables in different clauses are distinct even if they have the same names.

A clause can have several joint conditions or several alternative conclusions. Thus :

Grandparent(x, y) ← Parent(x, z), Parent(z,y)

Male(x), Female(x) ← Parent(x,y)

Ancestor(x, y) ← Parent(x,y)

Ancestor(x, y) ← Ancestor(x, z), Ancestor(z,y)

where x, y and z are variables, state that for all x, y, and z :

x is the grandparent of y if x is parent of z and z is parent of y;

x is male or x is female if x is parent of y;

x is ancestor of y if x is parent of y, and

x is ancestor of y if x is ancestor of z and z is ancestor of y.

Problems to be solved are represented by clauses which are denials.

The clauses

← Grandparent(Zeus, Harmonia)

← Ancestor(Zeus,x)

← Male(x), Ancestor(x, Dionisius)

where x is a variable state that :

Zeus is not grandparent of Harmonia,

for no x is Zeus ancestor of x, and

for no x is x male and is x an ancestor of Dionisius.

A typical problem-solver (or theorem-prover) reacts to a denial by using other clauses to try to refute the denial. If the denial contains variables, then it is possible to extract from the refutation the values of the variables which account for the refutation and represent a solution of the problem to be solved. In this example, different refutations of the second denial find different x's of which Zeus is the ancestor.

x = Ares, x = Harmonia, x = Dionisius.

Clausal form has the same expressive power as the standard formulation of predicate logic. All variables $x_1,\ldots,x_k$ which occur in a clause C are implicitly governed by universal quantifiers $\forall x_1, \ldots, \forall x_k$ (for all x_1 and ... and for all x_k). Thus C is an abbreviation for $\forall x_1 \ldots \forall x_k$ C.

The existential quantifier $\exists x$ (there exists an x) is avoided by using constant symbols or function symbols to name individuals. For example, the clauses

Father (dad(x), x) ← Human (x)

Mother (mum(x), x) ← Human (x)

state that for all humans x, there exists an individual, called dad who is father of x, and there exists an individual, called mum(x) who is mother of x.

If we restrict the clausal form so that at most one conclusion is allowed on the left hand side of the arrow we have the Horn clause subset of logic. Theorem provers exist that execute logic programs in Horn clause form with efficiency comparable to conventional language interpreters. Horn clauses are a relational version of the first order recursion equations introduced earlier. Thus the familiar factorial function becomes

(F1) Fact(o,s(o)) ←

(F2) Fact(s(x),u) ← Fact(x,v), Times(s(x),v,u)

Regard the terms o,s(o), s(s(o), ... as the numerals 0,1,2... and read Fact(x,y) as stating that the factorial of x is y and Times (x, y, z) that x times y is z. Given a program (or set of clauses) for computing the Times relation, (F1) and (F2) constitute a program for computing the factorial relation. To compute the factorial of 2 we add to the program the clause.

(F3) ← Fact (s(s(o)), x)

which states that no x is the factorial of 2. This contradicts (F1) and (F2) which logically imply that the factorial of 2 is 2. There exist efficient proof procedures which detect this contradiction and compute the factorial of 2.

The language PROLOG based on the procedural interpretation of Horn clauses was designed and implemented by Colmerauer and Roussel in Marseilles (Colmerauer et al 1975). A PROLOG compiler for the PDP 10 has been written that compares in efficiency with compiled pure LISP (Warren et al 1978). Similarly a PROLOG interpreter for the IBM 370 has been shown to compare with PASCAL for runtime efficiency (Roberts 1977).

3.1 Program Synthesis in logic

The fact that logic has a well defined model theory and proof semantics means that program transformation or synthesis can be easily carried out on logic programs. Any new theorem that can be derived from the statements of a logic program is valid in the sense that its incorporation into the program will not change the meaning of the program but may improve its behaviour.

The various forms of logic - standard form, unrestricted clausal form, and Horn clause form - represent a hierarchy of specification levels. As we descend this hierarchy expressive power is sacrificed to achieve more efficient interpretive theorem provers. Transitions between forms and re-arrangements within a form are (only!) logical deductions that cannot change a program's meaning but may dramatically increase its performance. Bibel (1976), Clark and Sickel (1971) and Clark and Darlington (1980), and Hogger (1979) show how transformation of logic programs can be achieved.

There is another dimension along which program transformation can be carried out on logic programs. A logic program will in general be non-deterministic and will be interpreted by a back-tracking theorem prover. Different control strategies will result in varying degrees of efficiency. Thus the performance of a given logic program can be improved, without its meaning being changed, by providing extra control information or annotations that act as advice to the theorem prover as to the best order to go about things. This thesis, that "algorithm = logic + control", is forcibly made by Kowalski (1979) and has been incorporated in the language IC-PROLOG (Clark and MacCabe 1979). The same idea of improving the behaviour of a program by providing control annotations has been applied to recursion equations by Schwartz (1977).

3.2 Logic Programming conclusion

Logic provides a flexible, very high level programming language. Because of its relational and non-deterministic nature it has many applications outside the range of conventional languages, for example as a data base query language and in expert systems. It has been applied to many large scale problems, particularly in Hungary where several important programs, including a drug interaction program, have been written in PROLOG.

I feel that the use of logic based programming languages is bound to grow and that they should be included in any computer science syllabus. Again it puts programming on a sound mathematical basis and provides links between the fields of programming, artificial intelligence and mathematical logic.

4. Conclusion

As I stated in the introduction the goal of totally automating the programmer's skills still lies far in the future. However unless one denies the possibility of producing artificial intelligence one must admit that it is an area worthy of continuing work. In the short term this work has produced useful insights into programming.

The approaches that I have concentrated on, transformation and logic programming, can both be characterised as attempting to put programming languages onto a sound mathematical footing. Of course many other approaches to program design share the same aims but the approaches I have outlined differ in using existing mathematical formalisms as languages and in providing a formal calculus in which the derivation of programs can be stated, offering the opportunity of machine processing. Whether these ideas will become widely adopted depends on many factors, some of them outside scientific consideration. However, certain trends are in their favour. Software is now produced in such a costly and unreliable way that the much heralded explosion in the use of computers will be disastrous unless we radically change our habits. The declining cost of hardware that is making such an expansion possible is also making parallel processing a possibility. The applicative languages are ideal for such implementations. In fact the data flow languages (for example, Arvind and Gostelow (1977) and Gurd and Watson (1978)) are either openly functional or imperative languages restricted to behave functionally. The declining cost of hardware also means that we can afford to be more liberal in providing systems to assist programming. Of course no amount of hardware can overcome a basically exponential process but I believe that dramatic advances in program reliability and useability are possible in a short time scale.

Discussion

Professor Katzenlson pointed out that a global analysis of a program is usually necessary in order to improve it. He wondered why there had been no mention of this technique in the improvement of NPL programs.

Dr. Darlington explained that NPL is a purely applicative language, and for that reason no global analysis is necessary. This is in contrast to SETL.

Professor Katzenelson then enquired whether any more had been done on optimising storage strategy in NPL programs. **Dr. Darlington** agreed that this would be a necessary final step to producing a good program. A complete analysis can be made within the recursion equation structure, though it has not been done yet. Again no global analysis would be required.

Professor Michaelson asked whether any large numerical problems had been tackled within the applicative framework. **Dr. Darlington** replied that he intended to, but had not done so yet.

Professor Dijkstra expressed his doubt about what can be achieved by transformation systems of the kind which Dr. Darlington had described; for example, whether such a system could derive the logarithmic form of the Fibonacci function. This doubt is based on the observation that the most efficient forms of algorithms often rely on mathematical theorems (sometimes rather deep theorems), and hence the transformation from a simple program to an efficient form must embody the power of a theorem prover. Even with external guidance the transformation path could be very long if it exists at all.

In reply **Dr. Darlington** stated that he believed such systems to be of practical value in program development, even if it were necessary for the programmer to provide advanced forms of guidance.

Professor Paul, quoting Dr. Darlington's remark that logic programming "unifies logic, programming and artificial intelligence", asked whether logic programming was likely to have any impact soon on general programming styles.

Dr. Darlington reported that programmers with Prolog experience claim a great change in their own style, and that they believe logic programming will have more widespread impact soon.

Dr. Henderson pointed out that recursion equation languages seem very unnatural to conventional programmers - and Prolog will seem even more so. This will present a problem for the spread of logic programming.

References

Arvind and Gostelow, K. (1977). "A computer capable of exchanging processing elements for time", Information Processing 77, B. Gilchrist (Ed) North Holland, New York, pp. 849-857.

Backus, J. (1978). "Can programming be liberated from the van Neumann style? A functional style and its algebra of programs", Turing Lecture, CACM 21, (8), pp. 613-641.

Bauer, F.L., Partsch, H., Pepper, P. and Wossner, H. (1977). "Notes on the project CIP: outline of a transformation system", TUM-INFO - 7729, Technische Universitat Munchen.

Bibel, W. (1976). "Syntheses of strategic definitions and their control", Bericht Nr. 7610, Abt. Mathem, Techn. Munchen.

Burstall, R.M. (1977). "Design considerations for a functional programming language", Proc. of Infotech State of the Art Conference, Copenhagen, pp. 45-57.

Burstall, R.M. and Darlington, J. (1977). "A transformation system for developing recursive programs". JACM Vol. 24, No. 1, pp. 44-67.

Burstall, R.M., MacQueen, D. and Sannella, D. (1980). "Hope, a Higher order applicative language", Report, Dept. of Computer Science, Edinburgh University.

Clark, K. and Sichel, S. (1977). "Predicate logic : a calculus for the formal derivation of programs", Proc. Int. Joint Conf. Artif. Intell.

Clark, K. and Darlington, J. (1980). "Algorithm classification through synthesis", Computer Journal 23, pp. 61-65.

Clark, K. and McCabe, F.G. (1979). "The control facilities of IC-PROLOG", Report, Dept. of Computing and Control, Imperial College, London.

Colmerauer, A., Kanoui, H., Pasevo, R. and Roussel, P. (1972). "Un systeme de communication homme-machine en francais", Rapport preliminaire, Groupe de Res. en Intell Artif. U. d'Aix-Marseille Luminy.

Darlington, J. (1975). "Application of Program Transformation to Program Synthesis", Proc. of International Symposium on Proving and Improving Programs, Arc et Senans, France, pp. 133-144.

Darlington, J. (1977). "Program transformation and synthesis : present capabilities", Report 77/43, Dept. of Computing, Imperial College, London.

Darlington, J. (1979). "The synthesis of implementations for abstract data types", Report, Dept. of Computing and Control Imperial College, London.

Darlington, J. and Feather, M.S. (1979). "A transformational approach to modification", Report, Dept. of Computing and Control, Imperial College, London.

Feather, M.S. (1979). "'ZAP' Program transformation system: Primer and User's manual", DAI Research Report No. 54. Edinburgh.

Gerhart, S.L., Lee, S. and deRoevar. W.P. (1979). "The evolution of list copying algorithms", Proc. 6th POPL, Texas, pp. 53-67.

Green, C.C. (1969). "Application of theorem proving to problem solving", Proc. Int. Joint Conf. on A.I., Washington D.C., pp. 219-239.

Gurd, J., Watson and Glauert, (1978). "A multi layered data flow computer architecture", Dept. of Computer Science Rept., Univ. of Manchester.

Guttag, J.V. (1977). "Abstract data types and the development of data structures", CACM 20 (6), pp. 306-464.

Hogger, C. (1977). "Deductive synthesis of logic programs", Res. report Dept. Computing and Control, Imperial College London.

Kernighan, B.W. and Plauger, P.J. (1976). "Software Tools", Addison-Wesley.

Kowalski, R.A. (1979). "Algorithm = logic + control", CACM 22, 7, pp. 424-436.

Manna, Z. and Waldinger, R. (1971). "Towards automatic program synthesis", CACM 14,3, pp. 151-165.

Manna, Z. and Waldinger, R., (1979). "Synthesis : dreams => programs", IEEE Trans. Softw. Eng. SE-5, pp. 294-328.

Manna, Z. and Waldinger, R. (1980). "A deductive approach to program synthesis", TOPLAS 2 (1), pp. 90-121.

Schmitz, L. (1978). "An exercise in program synthesis : algorithms for computing the transitive closure of a relation", Bericht 7801, Fachbereich Informatik, Hochschule der Bundeswehr, Munchen.

Schwartz, J. (1977). "Using annotations to make recursion equations behave", Res. Memo, Dept. of Art. Int., Edinburgh Univ.

Roberts, G. (1977). "An implementation of PROLOG", M.Sc. thesis. University of Waterloo.

Robinson, J.A. (1965). "A machine-oriented logic based on the resolution principle", JACM 12 (1), pp. 23-41.

Waldinger, R. (1969). "Constructing programs automatically", Ph.D. thesis, Carnegie-Mellon Univ.

Waldinger, R. and Lee, R.C.T. (1969). "PROW : a step toward automatic program writing", Proc. IJCAI, Washington, D.C.

Warren, D., Pereira, L.M. and Pereira, F. (1977). "PROLOG - the language and its implementation compared with LISP", Proc. Symp. on AI and Programming languages (SIGPLAN Notices).