The Competence/Performance Dichotomy in Programming
Preliminary Report

Vaughan R. Pratt
Massachusetts Institute of Technology
Cambridge, Mass., 02139

Abstract

We consider the problem of automating some of the duties of programmers. We take as our point of departure the claim that data management has been automated to the point where the programmer concerned only about the correctness (as opposed to the efficiency) of his program need not involve himself in any aspect of the storage allocation problem. We focus on what we feel is a sensible next step, the problem of automating aspects of control. To accomplish this we propose a definition of control based on a fact/heuristic dichotomy, a variation of Chomsky's competence/performance dichotomy. The dichotomy formalizes an idea originating with McCarthy and developed by Green, Hewitt, McDermott, Sussman, Hayes, Kowalski and others. It allows one to operate arbitrarily on the control component of a program without affecting the program's correctness, which is entirely the responsibility of the fact component. The immediate objectives of our research are to learn how to program keeping fact and control separate, and to identify those aspects of control amenable to automation.

1. Transferring Responsibility to the Computer.

One might characterize the difference between a tool and a servant (or assistant, to avoid anachronism) solely in terms of intelligent communication. A user commands his tool in the simple language of the tool; a master commands his assistant in the rich language of the master, putting a heavy demand on the assistant's intelligence. The modern computer began as a tool for calculating tables; a reasonable objective of artificial intelligence is to make it an intelligent assistant. In this paper we outline a step in this direction.

As an example of progress to date we may consider storage allocation, the problem of finding a place in memory to store a datum. The machine language programmer must specify an absolute address for an array. The symbolic assembly language programmer may simply request a given amount of storage at assembly time, leaving the choice of base address up to the assembler. The Fortran programmer can store data in two-dimensional arrays without needing to choose or even be aware of a correspondence between integers and pairs of integers for mapping two dimensions into one. The Algol programmer can have multiple simultaneous activations of the same procedure (namely when that procedure calls itself directly or indirectly, i.e. recursively) without having to allocate storage in advance for each such activation. The Lisp programmer can create complex objects (e.g. by appending lists) without having to declare them in advance to the storage allocator and without even having to think up names for them. We observe in this progression a gradual shift of responsibility for storage management from the programmer to the computer. Every such responsibility the computer can assume means that much less work for the programmer, and, more importantly in our opinion, that much less opportunity for clerical oversights on the part of the programmer.

One might take the position that such progress, while of value to programmers, is not a step towards an intelligent assistant, but merely an application of well-understood techniques from the theory of algorithms. We dispute that position; how responsibility is transferred to the computer, whether via well-understood algorithms or magic, is immaterial when it has been transferred. Our objective is to continue this transfer, hopefully without having to resort to magic.

In this paper we will focus not on data management but on control; our objective is to automate some of the control decisions currently made by the programmer. In the following we outline a philosophy of control in terms of interpreters and programs, present an approach to separating factual issues ("competence") from control or heuristic issues ("performance") in programs, and discuss approaches to automating the latter.

2. A Philosophy of Control

We adopt the viewpoint that an operating computer consists of an interpreter and a program being interpreted, and that the program guides/advises/constrains the interpreter in its choice of operations in much the same way as a grammar constrains a speaker. Readers unfamiliar with this viewpoint might consider the regular expression A:=1;(X>0;A:=XxA;X:=X-1)*;X=0, a grammar of sorts, where the assignments and tests are terminals, ";" is concatenation and "*" is Kleene closure; this grammar generates the set of execution sequences associated with one way of computing X! . Out of context, a grammar does not specify which of the sentences in its language the speaker should use, any more than does a program specify which execution sequence to use in the absence of an environment.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Given an environment, whether a "simple-minded" intepreter can choose an appropriate execution sequence depends on the nature of the program. In the factorial example, the interpreter can work "from left to right" in the regular expression, inspecting the tests X>0 and X=0 to decide whether to execute X>0;A:=XxA;X:=X-1 again or to end it all by executing X=0 , using the environment in carrying out the rule "never execute a test whose value is false." With this rule, the interpreter's options are so limited by this program that almost no intelligence is needed to choose the right sequence, in that the interpreter need not be a large program and need not consult other data bases beyond the immediate environments.

This viewpoint of the program as guide is not constrained to execution sequences of assignments and tests. The formulae f(0) = 1 and f(n+1) = (n+1)xf(n) can be regarded as advice on what to do with, say, f(3) , one obvious possibility being to rewrite it repeatedly using the above formulae and other facts of arithmetic to end up with f(3) = (2+1)xf(2) = 3xf(2) = ... = 6xf(0) = 6x1 = 6 . Again a simple interpreter will suffice to discover this sequence, say by trying to unify the last term generated with the left hand side of one of the above two identities, where the unifier has enough intelligence to unify 3 and n+1.

In the above examples the interpreter contributed next to nothing in the collaboration between program and interpreter. This was about the extent of the power of machine languages on all early computers. One might be tempted to justify this on the basis of the theoretical observation of Turing [21] that small universal interpreters exist, that is, that in the division of labor between program and interpreter, all but a small fixed amount of the labor can be allocated to the program. However, it has always been understood by language designers, whether consciously or subconsciously, that the savings in hardware possible using a minimal interpreter are more than offset by the resulting inefficiency of execution and inconvenience to the programmer, so that even the simplest machines use machine languages considerably more sophisticated than that implied by Turing's choice of representation of programs to be executed by his universal machine. With the rapidly decreasing ratio of hardware to programmer costs, and with the development of services provided by operating systems as a powerful extension of the basic machine language, this division of labor has been steadily shifting, with the interpreter assuming progressively more responsibility in the decision making process.

The primary beneficiary of this trend in the division of labor is the programmer, in his capacity as writer of his own programs, and as reader/modifier of both his own programs and those of others. The more intelligent the interpreter, the less advice the programmer need supply to enable the interpreter to accomplish a given task. It is natural to assume that less advice leads to less program-writing effort, but we prefer to de-emphasize this potential benefit and instead stress the notorious unreliability of the programmer in comparison to the computer. Less advice means less opportunity for programmer-generated error. If this results in a substantial decrease in debugging time, some increase in programming time may be tolerable. However, the greatest beneficiary in the end may be the program's user, for whom the cost of a bug in the program may often amount to more than the cost of the program, even if he is the sole purchaser of that program and has absorbed the full programming costs.

3. Separating fact and heuristic.

We focus on the notions of competence (or fact, or epistemology) and performance (or heuristic), which supply a dimension for modularizing programs that has received relatively little attention to date. This dimension originates with Chomsky [2], who argued the relevance of the competence/performance dichotomy to theoretical linguistics. The idea has implicitly been applied to computer science in various ways which we discuss later. The competence of a deterministic program is that information in the program that contributes to its partial correctness (soundness) and "non-deterministic termination" (completeness). The performance contributes to the determinism of the program. In this section we shall be concerned simply with making the separation, being content to leave the automation of the heuristic component till later. This is a little vague, so let us consider some examples.

The simplest domain in which to study the dichotomy is that of function definitions, where one combines identities with information as to when each identity is relevant. This can be observed in the following definition of the factorial function:

f(x) : if x=0 then 1 else x*f(x-1) .

We find here two kinds of information. One is factual: the definition asserts that f(0)=1 and that if x≠0 then f(x)=x*f(x-1). The other is procedural, telling the interpreter in what order to do things; the advice given here is to begin with the fact f(0)=1 , treating it as f(0)→1 (asymmetric =), and if it is inapplicable, to proceed to the other fact (with no provision for advice if the second fact is inapplicable, e.g. if the multiply hardware is broken).

If we were to rewrite this definition separating the competence from the performance, we might write:

Competence:
base: f(0) = 1 .
ind: f(x+1) = (x+1)*f(x) .

Performance:
integer(f(x)): LtoR(base)|LtoR(ind) .

The competence component supplies the two facts, appropriately labelled. The performance component says that if you want the expression f(x) transformed into an integer then apply the Left-to-Right transformation rule to f(x) using the identity "base," and failing that, try the same with the identity "ind." This use of labels is inessential; whether we write factorial as above or as

integer(f(x)): LtoR(<f(0)=1>) | LtoR(<f(x+1)=(x+1)*f(x)>)

is not important so long as we can distinguish the two components, in this case by putting the elements of the competence component in meta-brackets. With further syntactic sugaring one might be able to get the definition to look almost the same as the original one. However, there is an aspect of competence not yet discussed that makes it likely that labels would be widely used, and that is the likelihood that much of a program's competence component will consist of generally useful facts appealed to by more than one program. A trivial instance of this is the gcd example below.

Now that we have seen an example of the idea, let us return to definitions. We referred in the beginning to "soundness" and "completeness" as though the competence component were an axiom system. Indeed, we shall think of it as just that, without the usual stigma attached in axiom systems to non-independent axioms; that is, we shall not object in the least to the presence of axioms that follow from other axioms. The advantage of having a set of logically independent facts in the competence component is that it helps to keep that component small. The disadvantage is that it shifts to the interpreter and the performance component the burden of coming up with the relevant theorems. We feel that this task is more appropriate for whoever establishes the correctness component. In this way we keep separate the functions of generating the proof of the program's correctness and executing the program.

Our notion of "soundness" is here one of fidelity to the properties of the "problem domain." Thus no fact in the competence component should be inconsistent with what we know about the domain; being able to prove such facts is a sure guarantee of this. The two facts used in the factorial example, if not axiomatic, can at least be proved from whatever definition of factorial one has in mind.

Our notion of "completeness" is a bit more subtle. The basic idea is to make sure that there are "enough" facts around. Our objective is to make it possible for the interpreter and the performance component to run using at most the resources specified by the programmer, who may say simply "the program should halt on all inputs," or, more demandingly, "it should not take time greater than $\text{sqrt}(n)*\log^2(n)$." The competence component we gave for factorial is complete with respect to the first of these, but not the second, unless the interpreter does some clever theorem proving; some more facts are needed, such as

$n^2! = P(0)*P(n)*P(2*n)*...*P((n-1)*n)$
where $P(x) = (x+1)*(x+2)*...*(x+n)$

together with enough facts (or a library procedure) to allow one to evaluate the degree n polynomial P at n points in time $O(n \log^2 n)$ [1].

The following example illustrates issues absent from the simple factorial example.

```
define gcd(x,y) : if x=0 then y else gcd(y mod x, x) .     (Euclid)

define gcd(x,y) : if x=0 then y                            (J. Stein)
            else case [even(x),even(y)] of
            ([t,t]:  2*gcd(x/2,y/2),
             [t,f]:  gcd(x/2,y),
             [f,t]: gcd(y,x),
             [f,f]: if x≥y then gcd(x-y,y)
                       else gcd(x,y-x)) .
```

Separating out the components, we have:

Competence.

```
base: gcd(0,x) = x .
sym:  gcd(x,y) = gcd(y,x) .
add:  gcd(x,y) = gcd(x,x+y) .
com:  gcd(2*x,2*y) = 2*gcd(x,y) .
disj: gcd(2*x,2*y+1) = gcd(x,2*y+1) .
Euc:  gcd(x,y) = gcd(y mod x,x) .
```

(Note that there is no point in keeping separate competence components for separate programs.)

Performance.

```
(Euclid's algorithm)
integer(gcd(x,y)):
  LtoR(base)|LtoR(Euc) .

(Stein's algorithm)
integer(gcd(x,y)):
  LtoR(base)|LtoR(com)|LtoR(disj)|
  even(y)→LtoR(sym)|RtoL(add)|LtoR(sym) .
```

(Note the use of RtoL. Also note the use of the test "even(y)→ which causes "evenl(y) → L to R(sym)" to fail when y is not even." Tests in the performance component do not compromise any of the advantages of the competence/performance separation. In contrast, if we were to allow assignments, whether explicitly in the form of assignments to variables or implicitly by naming values to be returned as "the answer," we would then be able to compromise the correctness of the competence component.)

```
(Combined - useful when Euc is not known in advance to be
applicable, e.g. when availability of "mod" is unknown)
  LtoR(base)|LtoR(Euc)|LtoR(com)|LtoR(disj)|
  even(y)→LtoR(sym)|RtoL(add)|LtoR(sym) .
```

In this example we have more facts in the competence component than we need for either one of the two algorithms by itself. Conventional programming styles encourage "necessary and sufficient" programming, where you write down just enough cases to get the job done. In the competence/performance style, since the use of each fact is unspecified, there is not the same temptation to be "necessary," and one feels free to include any seemingly relevant facts. At the same time one runs the risk of not supplying sufficient facts; the "if-then-else" style forces one to cater for every case, whereas an empty competence component will at least be sound if not complete.

Impact of this Dichotomy on Verification Methods

Thanks primarily to the work of Floyd [4], we now understand the relationship between proving theorems in mathematics and proving that a program is correct, namely that correctness can be expressed in terms of theorems about programs, and that such theorems can be rigorously proved using inference rules analogous to rules used in conventional mathematics. Hoare [9] has made the connection even clearer by developing an axiom system modelled on systems developed by Hilbert and Gentzen. Manna [15] has shown that such systems are no stronger than conventional (non-program oriented) second-order logic, and Cook [3] has in effect strengthened this by showing that, even when the symbols of the underlying language are interpreted as arithmetic operations, such systems are no more powerful than first-order arithmetic (although his goal was actually to establish completeness of Hoare's method).

In the fact/heuristic style of programming, all of the above results become redundant. Correctness of a program reduces vacuously to correctness of the facts making up its competence component. The above results are now implicit in the interpreter, in which is vested the responsibility for "sticking to the facts." Thus all that is needed to prove the program correct is to prove the individual facts in the competence

component. In the example of a pay-roll program, verifying the facts becomes even simpler than proving mathematical theorems; it becomes a matter of having an accountant check that the facts are in accord with his understanding of the tax laws and related issues. (This observation is in accord with a hobby-horse of ours that truth transcends proof [18], that is, that the question of whether a formula is true arises independently of the existence of any axiom system in which it can be proved. Proofs are just computations for convincing mere machines limited by Church's Thesis of the truth of propositions. Correctness of a program is defined in terms of whether its facts are true, not whether they are provable. In this way we can factor out from our theory any reliance on the notion of proof.)

Procedural competence

Not all competence components need involve static facts. Sometimes a procedural idea is required, where the crux of the idea can be explained in terms of a small fixed number of steps of a procedure. Eliciting the appropriate control structure for application of the idea is still an appropriate task for the interpreter. In this case the programmer need only apply a small fragment of what we know about proving programs correct, just enough to verify that a couple of instructions have the desired effect; the interpreter can then be relied on to use the idea correctly. The competence components in the following examples refer to single-instruction programs.

In the following, symbols not taking arguments are considered universally quantified; all other symbols take on whatever value is assigned to them by the particular world in which they are evaluated. "P preserves Q" is an abbreviation for "P and Q implies next Q." "Next" is a unary modal logical connective (the only procedural concept present) whose meaning is "after executing one arm operation" (e.g. opening the hand, moving the hand so that B is caught (meaning that B is between the fingers of the hand, without the hand necessarily being closed), moving the hand to "at(B)" (meaning that the hand is hovering just over B).)

not on(b,b) $
not caught(table()) $
caught(b) and at(c) implies on(b,c) $
on(b,c) and on(b,d) implies c=d $
at(x) and at(y) implies x=y $

on(x,b) implies next not caught(b) $
on(x,c) preserves not on(b,c) $
on(b,c) implies next (open() implies on(b,c)) $
caught(b) implies next (open() or caught(b)) $
open() preserves on(b,c) $
not open() preserves caught(b) $
not open() preserves not caught(b) $

The reader familiar with interpreters of theorems (as in [5,6,7,10,12,20]) should not find it hard to see how to interpret these facts as a program. (The monkey-and-bananas problem of [12] is the canonical example of this phenomenon.) Three of the predicate symbols, "at", "caught" and "open", have the appropriate effects on the arm. Note the non-monotonic application of the first two facts, discussed in the next paragraph. We have not yet worked out a suitable heuristic component for this example or the next; we are at present trying to reconstruct it from an already existing deterministic (i.e. conventional) LISP program that embodies essentially these facts.

An important decision to make in deciding just what a competence component is is whether to treat the facts exactly as in logic, where the soundness of the whole system can be reduced to the soundness of the individual facts, or whether one is willing to let some facts "over-ride" others. For example, a program to operate a robot arm needs to know somehow that a block cannot be placed on itself. This will prevent it from attempting to obey a command such as "Put B on B." An obvious approach to conveying this information is to supply the "fact" "not(on(b,b))." But if the correctness of the whole competence component reduces to the correctness of its facts and if it is correct with the fact "not(on(b,b))," it must also be correct without the fact. It follows that any contribution made by this fact to the correctness of the whole cannot be considered in isolation from the whole; we must have lost what we will call "monotonicity," the ability to add facts without increasing the correctness of the program (or conversely, to select any subset of facts from a correct library knowing that the subset will also be correct).

In the following system we eliminate non-monotonicity, and also replace a single modality (next) with a separate modality for every elementary arm operation, represented for convenience here as assignments to the three arm predicates "at," "caught" and "open." [at(b):=t] is interpreted to mean "after making the formula at(b) true," while <at(b):=t> means ¬[at(b):=t]¬ , the dual notion. (A close analogue is ∀x, interpreted as "after setting x to a random value," and its dual ∃x = ¬∀x¬ .) Thus [s]P says that no matter what happens as a result (direct or indirect) of doing s, P will hold; <s>P promises that it is possible for P to hold as a result of s terminating; when P is just t (true), this asserts that s terminates.

[P:=t]P $
[P:=nil]¬P $
on(b,c) implies on(dep(c),c) $
otherthan(c) ≠ c $
caught(b) and at(c) implies on(b,c) $
not on(dep(b),b) and (not caught(b) or open())
    implies <at(b):=t>t $
not on(dep(b),b) and open() implies <caught(b):=t>t $
not open() and caught(b) implies [at(y):=t]caught(b) $
caught(b) implies [open():=nil]caught(b) $
not on(x,y) implies [open():=nil]not on(x,y) $
not on(x,y) implies [caught(b):=t]not on(x,y) $
on(b,otherthan(c)) implies not on(b,c) $

The reader can verify for himself that every assertion in this system is valid (given a reasonable interpretation of how the blocks world behaves) by itself, unlike the preceding system, where in the absence of such pessimistic facts as "not(on(b,b))" it would be possible to deduce a computation that put B on B.

It is worth noting that in neither of the above systems is the notion of state referred to explicitly. This is in contrast to [5] and [12]. See [18] for a rigorous account of how to do this while keeping the semantics intact.

These two examples suggested two ways in which procedural information might be incorporated into facts. This leads one to ask whether there is any limit to how much procedural information

can be put in facts. This depends on what programs one is willing to allow in modalities. The equivalences [18]

$$[a \cup b]P \equiv [a]P \wedge [b]P$$
$$[a;b]P \equiv [a][b]P$$

show that excluding union (i.e. non-deterministic choice) and sequencing does not affect what we can say, leading us to include them without any additional concern. Only when we add iteration need we again question whether our fact language has become too rich. Obviously one can then simply implement an interpreter that executes b when the fact component consists of just [b]P, announcing P when it terminates along with the current values of those variables affected by b. This is the phenomenon of "the richer the language, the more complex the ideas expressible in it." In a weaker language, it is up to the interpreter, with the help of the heuristic component, to figure out the complex ideas; the fact component can only contribute relatively simple ideas.

Other work.

Though we are unaware of any published work in computer science that explicitly draws attention to the distinction between competence and performance, the idea is implicit in a number of places. The most familiar application of the idea is in the wide-spread use of grammars in parsing systems. A grammar, whether for Algol or English, can be thought of as a collection of facts about a language. The context-free rule A::=BC may be read as the fact "An A may be a B concatenated with a C." From such facts one may infer "An S may be a 'THE' concatenated with a 'DOG' concatenated with a 'EATS'." The objective of a parser is to recover the trail of reasoning leading up to such an inference. For programming language grammars the performance component is essentially empty and the interpreter (i.e. parser) supplies all the performance. Until Woods's ATN system [22], the same held for natural language parsers, where the entire grammar resided in the context-free component and the parser knew nothing about the special properties of the particular grammar it was to work with. Woods "augmented" his context-free grammars (represented as "transition networks" to facilitate this augmentation) with a performance component that helped the parser decide what to do when. All major context-free based systems for natural language being actively worked on today, whether or not represented as transition networks, now have a substantial performance component. (Kuno's Harvard Predictive Analyzer [11], which has an empty performance component, is maintained at IBM Yorktown Heights, but it is not actively being developed or used.)

An early advocate of explicit representation of heuristics in addition to facts was McCarthy [12]. However, McCarthy's emphasis was on the fact component, and neither that early work nor more recent work paid a proportional amount of attention to heuristics. However, this was made up for in the robot-oriented programs of the late 60's, notably Green's QA3 system [5] and Hewitt's Planner system [7]. The former relied on a theorem prover as its interpreter, and had an empty performance component; this leap into the utopia of fully automated heuristics proved premature. However, its competence component, consisting of raw facts, precisely matches our own ideals for a competence component. Planner was more performance oriented; however, enough of the performance was mixed in with the competence, e.g. by distinguishing antecedent theorems from consequent theorems when these were both merely implications, and by identifying the competence notion of negation with the performance notion of unprovability, that we would not wish to class Planner as a paragon of the method we advocate. Planner's successors (Hewitt's PLANNER73/ACTORS/PLASMA [8] and McDermott and Sussman's CONNIVER [13]) have become sophisticated performance oriented systems with no distinguishable competence and performance components as we envisage them. What distinguishes CONNIVER from PLASMA is CONNIVER's more immediate commitment to an intelligent interpreter, though this intelligence does not extend significantly beyond implementing some good algorithms for handling environments flexibly. More recently McDermott has developed a markedly more intelligent interpreter, supported by a small theorem prover. However, the performance component remains inextricably intertwined with the competence.

J. Schwartz's notion of "rubble" programs [19], programs consisting of a rubble of loosely inter-related fragments, captures one aspect of competence components, namely their discreteness. Schwartz's thesis is that any program has such a rubble program as its prototype; the programmer begins by formulating the rubble program, then proceeds to optimize it, in the process organizing the rubble into a structured unit and at the same time converting operations on sets into operations on their elements via the appropriate generalization of "reduction in strength." Schwartz proposes to automate this optimization. In this, Schwartz comes as close as any to sharing our objectives. Two major differences are Schwartz's preoccupation with reducing the expense of naive set manipulation and his more ambitious goals concerning automation of control. We ignore optimizations possible by reduction in strength (formal differentiation) because the examples we work with cannot benefit from this idea and we expect to see many more examples in that category. We have relatively modest goals for automating control because it appears to us that complete automation is at present enormously difficult. Partial automation seems sensible but begs the question of "what part?" At present our understanding of control structures does not permit us to carve them up and distribute them between the programmer and the computer, which is why we propose to work out our fact/heuristic dichotomy in more detail to try to elicit a better understanding of the heuristic component.

Another place in which the idea can be found is in the most recent attempts to implement "logic as a programming language," notably those of Hayes [6] and Kowalski [10]. These come as close as any work to meeting our own objectives for separating competence and performance. Both of them have the advantage of inheriting a well-defined semantics from logic (as did QA3), providing reassurance that proofs of correctness will not be just optimistic symbol manipulation but will be supported by a Tarskian definition of truth, for which to our knowledge no trustworthy substitute has ever been satisfactorily worked out. Hayes pays more attention to the development of a performance component than does Kowalski, who prefers to rely on the interpreter, which he implements using SL-resolution. Tarnlund [20] has implemented a version of Kowalski's system that incorporates the missing performance component.

Marr and Poggio [16] propose applying a four-level version of the competence/performance dichotomy to biologically-related computation. The top two levels resemble our fact and heuristic levels respectively, with their other two levels corresponding respectively to functions and their

implementation with digital hardware. Discussions with Marr suggest however that their top level is really a set of independent axioms (cf. our inclusion of theorems in the fact component) while their next level is a hybrid of our fact and heuristic components.

4. Approaches to Automation.

As we have little experience to date in separating fact from control, we are not in a good position to predict what approaches we will take to automating control. The only idea we have had to date concerns the non-monotonic style of programming exemplified by our blocks world example. As this idea is moderately interesting, we sketch it here. It was suggested by a technique used in R. Weyhrauch's first-order logic proof checker.

We view each fact as a collection of optimists and pessimists. To do this we restructure each fact slightly: all logical connectives (ignoring modalities for the moment, and assuming that only unary and binary logical connectives exist) are replaced by $\wedge$, $\vee$ and $\neg$ (e.g. $P \supset Q$ becomes $\neg P \vee Q$) and then the $\neg$'s are moved down to the atomic formulae via de Morgan's laws. (We think of formulae as trees with vertices labelled with symbols and having the root at the top.) The result is a monotone expression (one containing only $\wedge$'s and $\vee$'s) whose leaves are literals (possibly-negated atomic formulae). The monotonicity plays an essential role as we shall see. Note that this restructuring does not result in the substantial increase of size generally associated with conversion to conjunctive or disjunctive normal form unless $\equiv$ or $\not\equiv$ occur. For example, $(A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G \wedge H) \vee (I \wedge J \wedge K \wedge L \wedge M \wedge N \wedge O \wedge P)$ is not altered by our transformation, but is expanded to 64 literals if converted to conjunctive normal form. In fact our transformation does no violence to the basic structure of the expression in the absence of $\equiv$ and $\not\equiv$, changing only the names of some of the binary logical connectives and relocating some negations.

If P is an atomic formula we take $\neg P$ as usual; however, if P is $\neg Q$ for some atomic formula Q, we take $\neg P$ to be Q. In this way we can say that for any literal P, P is an optimist for P and the literal $\neg P$ is a pessimist for P.

Given a fact consisting of the single literal P, any attempt to make P true must meet with instant success since the existence of P as a fact means that P is always true. Thus if P occurs as a goal, the fact P is an optimist for that goal. Conversely, $\neg P$ is a pessimist for that goal if it occurs as a fact, saying that P can never be achieved.

Given a fact of the form $P \vee Q$, the occurrence of P in this fact still acts as an optimist for a goal P (and as a pessimist for $\neg P$) because all the operators "above" P in the formula (namely $\vee$) are monotone and do not reverse the sense of P. Thus if P is set as a goal, $P \vee Q$ says nothing pessimistic about it but is optimistic in the sense that if Q were false then P would be forced to be true. We call $\neg Q$ the requisite of this optimist. If the goal is $\neg P$ then $P \vee Q$ is only pessimistic, and says that Q had better be true or $\neg P$ will not be achievable. We call Q the requisite of this pessimist. If the fact is $P \wedge Q$, then this is as optimistic as the fact P for the goal P, and as pessimistic as the fact P for the goal $\neg P$, and say that there are no requisites.

These ideas generalize to arbitrary formulae in the normal form described above, because of the monotonicity. Given a goal P, if an optimist P occurs in a fact, then the set of requisites for that optimist are the negations of the siblings in each of the $\vee$'s encountered going from the occurrence of the optimist to the root of the expression. For example, if the goal is D and the fact is $(((A \vee B) \vee (C \wedge (D \vee E))) \vee F) \wedge (G \vee (H \wedge I))$, then the requisites of the optimist D are (in the order encountered going up the expression) $\neg E$, $\neg(A \vee B)$, $\neg F$ and $\neg(G \vee (H \wedge I))$. Similarly if the goal is $\neg D$ then the requisites of the pessimist D are E, $A \vee B$, F and $G \vee (H \wedge I)$.

This leads to an algorithm for achieving a goal P when P is a literal. First determine the truth of P; if true we are done. Now satisfy some requisite of every pessimist for P. This may entail recursively setting requisites as subgoals. If for some pessimist it proves impossible to satisfy any requisite then abandon the attempt to satisfy P. Otherwise find an optimist every one of whose requisites can be satisfied; success or failure here translates into success or failure in satisfying P.

We implemented an interpreter based on this idea and tried it out on the non-monotonic blocks world fact component, with an empty performance component. The absence of other facts that might throw the system off the scent allowed our interpreter to proceed reasonably quickly to satisfy such goals as "on(B1,B2)" in simple situations. Unfortunately, the presence of modalities such as "next" complicates this approach, and the heuristics we added to the interpreter to deal with the modalities did not seem satisfactory. Our next project is to give some thought as to how best to deal with "next".

The above method may be generalized to deal with $\equiv$, which avoids having to expand it out and destroy the original expression structure (though this need at worst square the expression size [17], roughly), by treating occurrences of a literal below $\equiv$ as both a positive and a negative occurrence. Any segment of a path from that occurrence to the root, with the segment lying between consecutive $\equiv$'s, may be forced to take on either truth-value, with the siblings at the two $\equiv$'s determining how that value interacts with the rest of the path. We leave the details to the reader.

Acknowledgments.

David Marr provided some stimulating argument. Richard Weyhrauch gave me the idea for section 4.

References.

[1] Aho, A.V., J. E. Hopcroft and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass. 1974.

[2] Chomsky, N. Some Aspects of the Theory of Syntax, MIT Press, Cambridge Mass. 1965.

[3] Cook, S.A. Axiomatic and Interpretive Semantics for an Algol Fragment. TR-79, Toronto, Feb. 1975.

[4] Floyd, R.W. Assigning Meanings to Programs. in Mathematical Aspects of Computer Science (ed. J.T. Schwartz), 19-32. 1967.

[5] Green, C. Cordell. The Application of Theorem Proving to Question-Answering Systems. Stanford University Computer Science Department Report CS-138. 1969.

[6] Hayes, Patrick J. "Computation and Deduction," Proc. Symp. Math. Found. of Comp. Sci., Czech Acad. of Sciences. 1973.

[7] Hewitt, C.E. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, MIT AI Lab TR-258. 1972.

[8] --------, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence," Proc. IJCAI 3, p. 235. 1973.

[9] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM 12, 576-580. 1969.

[10] Kowalski, R. Predicate Logic as Programming Language, University of Edinburgh Department of Computational Logic Memo 70. 1973.

[11] Kuno, S. The Predictive Analyzer and a Path Elimination Technique. CACM 8, 7, 453-462. 1965.

[12] McCarthy, J. Programs with Common Sense. In Semantic Information Processing (ed. Minsky, M.L.), MIT Press, Cambridge, Mass. 1968.

[13] McDermott, D. and G.J. Sussman. The Conniver Reference Manual, MIT AI Lab Memo 259a. 1972.

[14] --------. Flexibility and Efficiency in a Computer Program for Designing Circuits. Ph.D. Thesis, MIT AI Lab. Sept. 1976.

[15] Manna, Z. Second-order Mathematical Theory of Computation. Proc. 2nd Ann. ACM Symp. on Theory of Computation, 158-168. May, 1970.

[16] Marr, D.C. and T.G. Poggio. From Understanding Computation to Understanding Neural Circuitry. MIT AI Memo 357. May 1976.

[17] Pratt, V.R. Effect of Basis on the Size of Boolean Expressions. Proc. 16th Ann. IEEE Symp. on Foundations of Comp. Sci., 119-121. 1975.

[18] Pratt, V.R. Semantical Considerations on Floyd-Hoare Logic. Proc. 17th Ann. IEEE Symp. on Foundations of Comp. Sci., 109-121. 1976.

[19] Schwartz, J. A view of Program Genesis and its Implications for Future Programming Languages. In New Directions in Algorithmic Languages (ed. Schuman, S.A.), IRIA. 1976.

[20] Tarnlund, Sten-Ake. "An Interpreter for the Programming Language Predicate Logic," Proc. IJCAI 4, p. 601. 1975.

[21] Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem, Proc. London Math. Soc., ser. 2, 42, 230-265, 43, 544-546. 1936.

[22] Woods, W. A. Augmented Transition Networks for Natural Language Analysis. Report No CS-1 to the NSF, Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts. 1969.