EXPLOITING THE PROPERTIES OF FUNCTIONS TO CONTROL SEARCH

by

Alan Bundy

D.A.I. Research Report No. 45

October, 1977

EXPLOITING THE PROPERTIES OF FUNCTIONS TO CONTROL SEARCH

Alan Bundy

abstract

In the first half (sections 1-5) of the paper we examine the tradeoff between representing knowledge using functions and using relations. The properties of functions turn out to be indispensable, but to be a major contribution to the combinatorial explosion. In the second half (sections 6-12) an examination of these properties suggests incorporating them in the inference mechanism, where they can be used to great advantage in controlling search and reducing the combinatorial explosion. This incorporation is seen as the first step in the design of a computational logic attuned to the demands of automatic reasoning.

The purpose of the first half is introductory and may be omitted by those steeped in the traditions of Resolution theorem proving.

Contents

Exploiting the Properties of Functions to Control Search

1. Introduction

Several people in Artificial Intelligence have taken advantage of the fact that functions can be replaced by relations i.e. all expressions of the form

$$f(x_1, \ldots, x_n) = y$$

can be replaced by

$$P_f(x_1, \ldots, x_n, y)$$

Our first experience of this technique was in the work of Wos and Robinson 1965, where the group function + was replaced by a relation P i.e. $x+y=z$ was written $P(x,y,z)$. However, examples can be found all over the literature both explicit and implicit e.g. in Gelernter 1963, Bundy 1973, in STRIPS (see Fikes and Nilsson 1971), and in many uses of semantic nets. We understand from Jerry Schwarz that it goes way back to the thirties in mathematical logic

Since this technique is so widespread, two questions arise

(a) Is anything to be gained by it?

(b) Is there a price to be paid?

This paper addresses itself to answering these questions.

2. Existence and Uniqueness

We can start our investigation by asking "in what way do functions differ from relations". The answer in predicate calculus is quite simple. Functions carry with them two extra properties, namely:

(a) The existence of their value is guaranteed

(b) The uniqueness of their value is implied

Thus if I choose to represent the motherhood concept with a function, mumof(x), then I ensure that every individual has one and only one mother. e.g. If John is an individual then the existence of another individual who is his mother is guaranteed because mumof(John) is a term and Tarskian semantics demands that this term denote some individual. It is not possible to assert that John has two different mothers using only this notation.

If on the other hand I represent motherhood with a relation, Mother(x,y), then I get no such guarantees. It is quite possible that there is no individual a for which

Mother(John,a) is true

On the other hand we could easily assert that John has several mothers

e.g. Mother(John,Mary)

Mother(John,Sue)... where Mary $\neq$ Sue

Notice that this would be the correct situation for a concept like LOVES. It is quite in order for

LOVES(John,Mary)

and LOVES(John,Sue) to both be true; and there is no reason why John should love anybody.

If we have both the relation Mother and the function mumof in our system then we will have to relate them by some suitable axiom like

$\text{Mother}(x,y) \leftrightarrow \text{mumof}(x) = y$ (i)

In this theory the existence and uniqueness of John's mother can be quickly proved.

<u>existence</u>

mumof(John) = mumof(John)	by reflexive law of equality
Mother(John,mumof(John))	by (i) and modus ponens
$\exists$y Mother(John,y)	by $\exists$ introduction

<u>uniqueness</u>

→ Mother(John,x) & Mother(John,y)	assumption
mumof(John)=x & mumof(John)=y	by (i) and modus ponens
x = y	by transitively of equality

Mother(John,x) & Mother(John,y) → x = y by conditional proof

So just having the mumof function ensures that every individual has one and only one mother.

On the face of it then, it would seem that some concepts, like motherhood, are better represented as functions and some, like love, are better represented as relations. If the wrong choice of representation is made then either some vital information will be lost (e.g. everyone has a mother) or something untrue will be asserted (e.g. everyone has a lover).

3. A Decision Procedure

So if a function is replaced by a relation something is lost. How important is this loss? Is anything gained in return? A partial answer to these questions can be found by considering the logical properties of a theory containing no functions.

We will show that a theory containing no functions is decidable, i.e. there is a procedure for deciding whether a formula is a theorem or not and this procedure is guaranteed to terminate. This result was first pointed out to me by Bob Kowalski. For theories containing functions it can be shown that no such decision procedure exists. In this proof and for the rest of this paper the notions of functions and constants will be considered disjoint i.e. constants are not o-ary functions, functions are n-ary for 1.

The existence of this decision procedure establishes two things. Firstly, we cannot, in general, expect to do without functions. Secondly, on those occasions when we can manage without functions, it is best to do so as pleasant consequences follow from our abstinence.

We will investigate the decision procedure using the terminology of resolution theorem proving, because the investigative tools are better developed there. However, similar results could be established for other inference systems e.g. PLANNER type languages or semantic net inferencers.

In resolution theorem proving the theorems are proved by putting the axioms of the theory and the negation of the candidate theorem in clausal form. The resulting set of clauses S is fed to a theorem prover which tries to show that the set S is unsatisfiable. The theorem follows from the axioms if and only if S is unsatisfiable. It is this set S which we will require not to contain functions (except constants). This means that the original axioms and candidate theorem must not only be function free, but they must also be free of certain quantifier configurations, namely those that would give rise to functions under skolemization. The condition is that:

(a) in the axioms, existential quantifiers must not be preceded by universal quantifiers (when the axioms are closed) e.g. $\forall x \exists y\ P(x,y)$ is not allowed as it skolemizes to $P(x,f(x))$

(b) in the candidate theorem, universal quantifiers must not be preceded by existence quantifiers e.g. $\exists x\ \forall y\ P(x,y)$ is not allowed as, when negated, it skolemizes to $\neg P(x,f(x))$

We will show that if S contains no function there is a decision procedure which decides whether S is unsatisfiable. We will make use of Herbrand's theorem (see Kowalski and Hayes 1971) which states that:

"A set of clauses S is unsatisfiable iff there exists a finite contradictory set S' of ground instances of clauses in S."

A ground instance of a clause C is some substitution instance ($C\sigma$ say) of C, which contains no variable. The ground terms which replace the variables in C are formed only from functions and constants mentioned in the original set of clauses. If there are no functions or constants in S, then a new constant (a say) is invented and used to replace all variables.

If the clauses contain no functions (apart from constants) then there are only a finite number of ground instances of clauses in S. Thus we can enumerate all the contradictory sets S' and examine them for contradictions in turn. This examination for contradiction can be done by standard truth table methods and is guaranteed to terminate.

To see that there are only a finite number of ground instances of each clause C, we need to consider the possible substitution, σ, that can be applied to C. Since $C\sigma$ must contain no variables, σ must replace each variable with some ground term. If functions were available there would be an infinite collection of such terms

e.g. a, f(a), f(f(a)), f(f(f(a))),....etc where a is a constant and f a function

Without functions, however, there are only the constants to serve as ground terms. So there are only a finite number of essentially different substitutions, σ, which will yield ground terms and thus only a finite number of ground instances.

So the decision algorithm is:

(i) For each clause C in S form all the ground instances $C\sigma$ by replacing variables by constants in all possible ways.

(ii) Form all finite sets S' of ground instances.

(iii) Test each set S' for contradiction until either a contradiction set is found (terminate with success) or all sets are exhausted (terminate with failure).

4. Constrained Forward Inference

Of course this decision procedure is not particularly efficient and we might hope to do better. Plotkin has shown that we can get a very efficient algorithm if all the clauses are Horn clauses (see Welham 1976, App 3). A simple forward inferencing algorithm can be used which grows a search space polynomial in the original number of constants.

Plotkin's algorithm (which is basically just unit resolution) is:

Let Horn clauses be represented as

$$A_1 \; \& \; .. \; \& \; A_n \rightarrow B$$

$$A_1 \; \& \; .. \; \& \; A_n \rightarrow$$

where A_1 and B are positive literals and $n \leqslant o$

(i) Set DB to nil.

(ii If there is a clause of form

A_1 & .. & $A_n \rightarrow$

for which A_1 , .. , A_n match literals in DB with substitution σ then quit with success.

iii) If there is a clause of form

A_1 & .. & $A_n \rightarrow B$

for which A_1 , .. , A_n match literals in DB with substitution σ and if $B\sigma$ is not an instance of something already in DB then add $B\sigma$ to DB and go to (ii).

(iv) quit with failure.

This algorithm is guaranteed to quite with success if the original set of clauses is unsatisfiable. The reason is that we cannot go on adding to DB indefinitely, because there are only a limited number of different literals to add. On the other hand, the algorithm is known to be complete So we cannot continue to execute Step (iii) for ever but must quit at (ii) or (iii

Furthermore, the length of DB and therefore the number of iterations round the loop can never exceed

$$M.N^k$$

where M is the number of relation symbols

k is the maximum arity of any relation symbol

and N is the number of constants

The algorithm can be modified to act as a proof finder rather than refutation finder. The original set of clauses could consist of the axioms of a theory,plus the hypothesis of the theorem and we can test at each stage to see whether the conclusion of the theorem is true in DB.

Plotkin's algorithm has been used in spirit (albeit unconsciously) by a number of successful theorem provers, cf for instance Nevins 1974, Bundy 1973 and Ballentyne and Bennett 1973. These theorem provers made forward inferences from the hypothesis of the candidate theorem while keeping a lookout to see whether the conclusion of the candidate theorem was proved. Because forward inference is in general explosive these theorem provers employed various constraints on the inferences which could be made. These took the form of limits on the terms which could be introduced by an inference and corresponded to a ban on the use of functions to make new terms from old. Thus, even though the formulae contained functions, these were not used in an essentiallv "function-like" way and could have been replaced by relations without effecting the inferencing being done. For instance, in Nevins geometry program no new points could be introduced, except those mentioned in the original statement of the problem. There are only a finite number of configurations (triangles, parallel lines, etc.) between a finite number of points and these can all be represented by a set of relations between them. Functions are only indispensable for creating new points, e.g. as the intersection between non-parallel lines.

The surprising thing about the theorem provers of Nevins, Bundy and Ballentyne and Bennett is not that they were relatively fast and guaranteed to terminate. This much could be expected from the previous arguments about Plotkin's algorithm. The surprising thing is that they proved interesting theorems. Many simple theorems can be proved by this technique, i.e. without an essential use of functions.

5. An Embarrassment of Terms

We have seen the connection between functions and existence. A function guarantees its value will exist (section 2). An existentially quantified variable becomes a function on skolemization (section 3) e.g. $\forall x\ \exists y\ P(x,y)$ skolemizes to $P(x, f(x))$

In fact in clausal form one of the main uses of functions is to create new terms and thus introduce new entities into the argument. For instance, the function mumof(x) can be used to introduce an infinite string of new mothers (stretching back beyond Eve). Consider the formula

Human(x) → Human(mumof(x)) (ii)

loosely translated as "every human has a human mother". Used in forward inference mode and given the assertion

Human(John)

(ii) will deduce the assertions

Human(mumof(John))

Human(mumof(mumof(John)))

Human(mumof(mumof(mumof(John))))

etc

in rapid succession.

Clearly this is an embarrassment and needs controlling. The theorem provers discussed in section 4 control this explosion of new terms in an over-harsh way, by not allowing the creation of terms not already mentioned in the statement of the candidate theorem. Thus, they would not deduce Human(mumof(John)) from Human(John) unless mumof(John) appeared in the candidate theorem.

Unfortunately some theorems require the creation of new terms in their proof. What is needed is a compromise between the extremes of creating all possible new terms and creating none. The traditional resolution theorem proving technique for achieving this compromise was to impose an arbitrary function nesting bound. Thus, if the bound were 7, say, then terms up to

mumof(mumof(mumof(mumof(mumof(mumof(mumof(John)))))))

would be allowed, but longer terms would not. Clearly this is crude and unacceptable except as a short term solution. What is needed is to bring the creation of new terms under program control, so that the decision whether or not to create a term can be the subject of a complex decision making process.

As a first step towards providing such a facility, note that the explosive properties of formula (ii) disappear if mumof is replaced by the relation Mother. On translation (ii) becomes

Human(x) & Mother(x,y) → Human(y) (iii)

Now (iii) cannot be used in forwards mode to deduce Human(y) unless both Human(x) and Mother(x,y) are satisfied and Mother(x,y) will not be satisfied unless the mother of x is already known. To recapture the explosive properties of (iii) we would have to ensure somehow that Mother(x,y) was always satisfiable, a suitable y being created. The obvious way to do this would be to have an assertion of the form

Mother(x,mumof(x)) (iv)

which Mother(x,y) could match.

To achieve our compromise (iv) needs to be tempered with control advice about when the creation of the new term is to be allowed.

e.g. Allowed(mumof(x)) → Mother(x,mumof(x)) (v)

Now (v) can be used backwards to satisfy Mother(x,y) if the test Allowed(mumof(x) is passed.

What information might "Allowed(x)" use, to decide whether creation of term "x" is to be allowed? "Allowed" might, for instance, investigate the current state of the proof or the form of "x". In order to imitate the old function nesting bound, "Allowed" could be defined to count the degree of nesting in "x" and fail if this exceeded some threshold. Of course, in this case "Allowed" is really investigating the syntax of x, rather than its meaning, so to be strictly kosher "x" should be surrounded by Quine corner quotes, i.e. Allowed(⌜x⌝).

In the sections which follow we will be describing a program written in a "predicate calculus" like programming language, PROLOG (Warren 1977). In PROLOG it is possible to write predicates which investigate the syntax of their arguments as well as re-direct the search and other strictly non-kosher things. We will use the same notation for our program as we have used for predicate calculus formula so far, except that odd, non-predicate calculus things will start appearing in the clauses. We will try to explain these as they occur.

6. Term Creation in Equation Extraction

In our mechanics project (Bundy et al 1976) we were faced with just this need for controlled term creation in the process of equation extraction. Forming a particular equation often necessitates the formation of new intermediate unknowns, in addition to those already given in the statement of the problem. All these unknowns are represented in the program as constants. So the decision to be taken is whether this particular equation is wanted badly enough to justify the introduction of a new constant.

Our solution to this problem is as follows. At any stage in the process we have a list of sought unknowns and a list of givens. An unknown is taken from its list and a short list of candidate equations is formed on the basis of what kind of unknown we are trying to solve for and what situation it is defined in. We try first to find a candidate equation which will solve for the sought unknown without introducing any new intermediate unknowns. Only if this fails will intermediate unknowns be created.

This is implemented by having our equivalent of the "Allowed" function access a global flag. This flag is turned off during the first pass when intermediate unknowns are not being tolerated, is switched on for a second pass when intermediate unknowns have been shown to be necessary and is switched off again when a suitable equation has been formed.

The equation forming mechanism consists of a series of inference rules representing physical formulae, i.e. the formula for the constant acceleration formula $v = u+at$ is

Constaccel(object,period) &

CC(Accel(object,a,dir,period)) &

CC(Direction(period,t)) &

CC(Initial(period,begin)) & CC(Vel(object,u,dir,begin))

CC(Final(period,end)) & CC(Vel(object, v ,dir,end))

→ Isformula(v =u+a.t,constaccel-1,period-object)

This is called in backwards reasoning mode (as are all clauses in PROLOG) with the name of the formula, constaccel-1, and the situation, period-object, both bound, but with a variable for the equation. The satisfaction of each of the conditions, e.g. CC(Accel(object,a,dir,period)) fills in the details of the equation.

The "CC(....)" predicate is a special kind of call which serves the role of "Allowed(x)". "CC" stands for "creative call". It calls its argument in the normal way, but if this should fail and the global flag is on then appropriate new constants will be created. Thus,

CC(Initial(period,begin))

can create a new moment, begin, the initial moment of period and

CC(Vel(object,u,dir,begin))

can create a new velocity, u, in a new direction dir. Note that "CC" is used to create both new intermediate unknowns, like u, and other constants, like begin.

Sometimes we do not want a new constant to be created under any circumstances, so the "CC" will be omitted. For instance, one of the ways that constaccel(object,period) can work is to find the accel of object in period and see if this is invarient. There is no point in creating a new intermediate unknown for the acceleration as we will know nothing about it, including whether it is invarient. So the inference rule incorporating this knowledge is

NCC(Accel(object,a,dir,period)) & Invarient(a)

→ Constaccel(object,period).

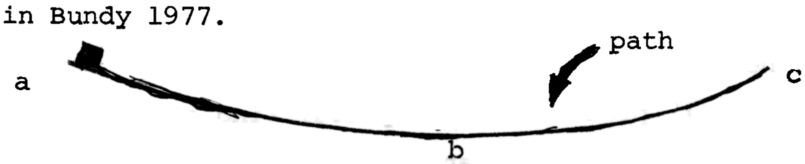
The NCC(...) predicate round Accel(...) behaves in a similar way to CC(...) except that it does not allow the creation of new constants. It is more fully explained in section 8.

Probably we will eventually have to think about more sophisticated controls over creation then these. But at least the provision of the creative call mechanism has enabled us to think clearly about the possibility of such controls.

7. Using Uniqueness to Prune Search

The uniqueness property of functions is not a potential source of explosion like the existence property. Rather it is an untapped source of control. Normally it is embodied implicitly in the equality axioms and not brought into play when needed (compare uniqueness proof in section 2). We will suggest extracting it from the equality axioms and incorporating it in the inference mechanism, where it can be brought fully into play.

Consider the following situation from the roller coaster world described in Bundy 1977.



Block is known to be at point a of path at moment begin. This is described by

At(block,a,begin)

Suppose the program at some stage tries to prove (and it will) that the block is at a point b at moment begin, i.e. tries to prove At(block,b,begin). Now we know this is silly, but the program doesn't and will try endlessly to prove the unprovable. What is needed is a trap to catch silly calls like this and reject them outright.

The situation is a general one. For instance, we would also like to reject an attempt to prove Initial(period,end) if we already know Initial(period,begin) and an attempt to prove Mother(John,Sue) if we already know Mother(John,Mary). The generalization is realized by noticing that At, Initial and Mother are all functions:

At from object X lines to points

Initial from periods to moments

Mother from animals to females

The inference mechanism must now be made to keep a look out for functions with all their arguments bound and see if contradictory information is already known.

The trap will not be appropriate for relations, e.g. it is quite alright to try to prove Loves(John,Sue) even though we already know Loves(John,Mary). Nor will it be appropriate if some of the functions arguments are unbound, e.g. if x is a variable it is quite alright to try to prove At(block,b,x) even though we know At(block,a,begin) since x will turn out to be some different time.

The uniqueness property can be used in at least one other place, namely to prevent backup when a function has been calculated. Consider the situation:

At(block,x,begin) & ....

Suppose At(Block,x,begin) is called and x is bound to a, but later processing of .... fails. Should processing backup and recalculate At(block,x,begin)? Any other answer is bound to be equivalent and lead to a similar failure of ...., since all properties of the new answer can be shown to be properties of the original answer. So backtracking can be killed in this case. Backtracking would be in order if the situation was

Loves(John,x) &

or

At(block,a,y) & ....

since recalculation in these cases could lead to a genuinely different answer.

8. The Inference Mechanism

The considerations of sections 6 and 7 have led us to design an inference mechanism for the mechanics project, incorporating controlled creation of new constants and exploitation of uniqueness information. Because of the provision of control primatives it proved possible to build the inference mechanism in PROLOG so that the existing simple depth first search was modified.

The inference mechanism we have built is certainly more widely applicable than the mechanics domain, since it relies only on general distinctions, e.g. function/relation rather than on special properties of mechanics. At the moment it is rather closely tied to depth first search, but this does not appear to be a crucial link and we hope to make it independent of the search strategy eventually.

In the new inference system there are no longer any functions, except in a few special cases i.e. in algebraic expressions and for representing sets. Functions are dispensed with because the extra properties they bring are incorporated in the inference mechanism. Instead, there are predicates, variables and constants. Predicates are marked to show whether they have the function properties or not, e.g. At(object,point,time) is a function from object X times to points, whereas Constvel(object,time) is relation between objects and times. In future we will use the terminology function/relation to describe this distinction.

A goal G is set up by calling the procedure CC(G) for creative call or NCC(G) for non-creative call. The difference is that whereas the CC call may ultimately result in the creation of a new constant the NCC call will not. The goal G is then analysed along two further dimensions to see: whether it is ground or general and whether it is a function or relation call. Ground means the goal is variable free, general means it contains variables. A function call means not only that the predicate has the function properties, but that its function arguments are bound

e.g. At(Block,x,begin) is a function call

but At(Block,a,y) is a relation call

and Constvel(Block,period1) is also a relation call

At this stage ground, function calls are checked to see if they contradict information which is already known, e.g. At(Block,b,begin) is rejected if At(Block,a,begin) is already known.

An attempt is then made to prove G (which may call the whole process recursively on subgoals). The goal may be satisfied by simple database look-up or by inference or it may fail. Different things then happen according to how the inference has been classified. There are 24 different classifications (i.e. 2x2x2x3), but generalizations enable us (and the program) to describe the different treatments succinctly.

Function calls and ground relation calls have further backtracking stopped, since further calls could only produce equivalent answers. Creative, general function calls which have failed, cause new constants to be created to fill their function value slots, provided a global flag is on, allowing new creations. Goals which have generated inferences are recorded in the database (either as successes or failures) so that these inferences will be short circuited if the same goals are ever called again

The replacement of explicit function notation by the special marking of predicates was motivated by design considerations in building the above mechanism. However, it brings with it a number of incidental representational advantages which we list below.

(i) A relation can be a function in more than one sense, e.g.

Timesys(period,initial,find)

a relation between a period and its initial and final moments can be a function from: its 1st argument to its 2nd; its 1st argument to its 3rd and its 2nd and 3rd argument to its 1st.

(ii) The inference mechanism could easily be modified to allow a predicate to have only some of the function properties.

e.g. uniqueness without existence, like the Godfather relationship
existence without uniqueness, like the ancestor relationship
existence with a modified uniqueness allowing a definite number of values, like the parent relationship

These concepts are messy to represent in conventional predicate calculus. We may be forced to make these modifications to deal with the motion predicates which describes motion of an object on a path during a period of time. Given the object and the time period the path is uniquely determined. However, it is not guaranteed to exist, since the object may not be in motion. Extending the notion of motion to include the degenerate case of stationary objects would create its own problems. At the moment motion is treated as a special case, but the discovery of similar predicates could motivate the proposed modifications.

9. Some Compromises

SILLY is the PROLOG predicate used to trap contradictory calls using the uniqueness property of functions, e.g. If At(block,a,begin) is known then SILLY(At(block,b,begin)) will succeed and the goal At(block,b,begin) will fail. In this section we will discuss some compromises used to implement SILLY and the consequences of them.

SILLY works by investigating the goal it is given and separating the function arguments from the function value. The function arguments of At(Block,b,begin) are Block and begin, and the function value is b. A variable is then substituted for the function value and SILLY looks in the database to see if it has a different value stored, e.g. At(Block,x,begin) is used as a pattern to search the database, which binds x to a. The previously stored value is then compared to the new one to see if they are different, e.g. a is compared to b.

The compromises in this procedure are:

(i) Only database look-up is used to find contradictory values, instead of arbitrary inferencing.

(ii) The difference check only looks to see if the constants are spelt differently, rather than denoting different entities.

Either of these could be easily replaced. Database lookup by some limited inference call, or even by a co-routine working in harness with the attempt to prove the original goal. Different spelling by some other cheap method of proving things different,or again a co-routine.

Since database lookup is a bit lapse in spotting contradictory calls it may let something pass which should have been stamped on. On the other hand, different spelling is overkeen and may stamp on something which should have been passed. Neither of these is fatal to the inference system in that soundness will be preserved in either case.

10. The Unique Name Assumption

The assumptions behind the different spelling procedure could do with further exploration. Basically it is founded on the belief that two things are different if they have different names - the unique name assumption.

In a conventional predicate calculus system this would be far more dangerous than it is here. With the normal proliferation of terms it is quite easy to give the same thing two names. Consider the situation of two consecutive periods of time sharing a common moment

period 1 period 2

shared moment

This shared moment might be described as initial(period2) and final(period1).

Examination of the inference mechanism described in section 8 will show that we are very conservative about the provision of new names. A new name is only created after every attempt has been made to show that a suitable one does not already exist. Suppose that "bliss" was the name of the shared moment and that

Initial(period2,bliss)

was already known. A request for

GC(Final(period1, x))

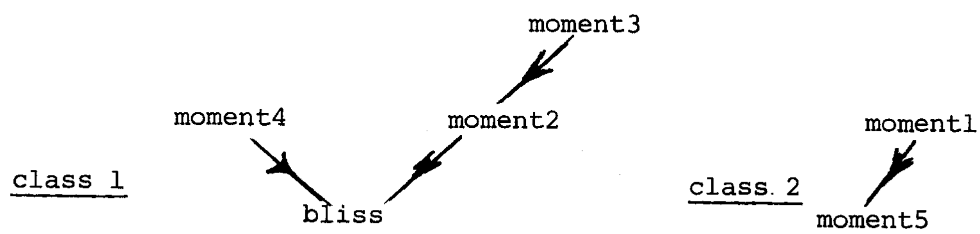
would start by trying to satisfy

Final(period1, x)

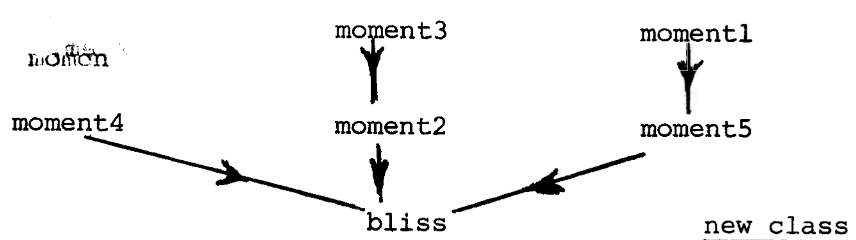
which, by a process of inference would succeed binding x to bliss. Creation of a new name for x would now be prevented.

Unfortunately, this mechanism is not fool-proof. Imagine a situation in which new information was being fed to the program as problem solving was taking place. Suppose that the program did not yet know that period1 and period2 were consecutive. Then the above attempt to satisfy Final(period, x) would fail and a new name, e.g. moment1 would be created. If at a later stage the information that period1 and period2 were consecutive were input then the program would have two names, moment1 and bliss, for the same moment.

This situation has not yet arisen in the mechanics project because all information about a problem is input at the beginning before problem solving takes place and all our problem solving is done by backwards inference. When the natural language understanding and equation extraction sections of our project are properly integrated it will become a problem, so we propose to adopt the following situation. Each object will belong to an equivalence class, organised as a tree, with the root as distinguished member. e.g.



We hope that these equivalence classes will normally contain only one element. When new information is input this will cause a process of forwards inference. For instance, when period1 and period2 are asserted to be consecutive, the process of forward inference would establish that bliss and moment1 name the same moment. This would cause the two equivalence classes to be joined, by pointing the root of one to the other. e.g.



A new difference check would be written to return true iff x and y belong to different equivalence classes.

11. A positive use of Uniqueness

Both the uses we have made so far of the uniqueness property have been negative, i.e. pruning the search space of fruitless paths. But it is also possible to use uniqueness positively, to help prove something, albeit a negative fact. For instance, if we know:

At(Block,a,begin)

then we also know

Not(At(Block,b,begin)) since $a \neq b$.

This information is probably best incorporated not in the inference mechanism, but as a rule about Not. i.e. If G is ground and SILLY(G) (in the sense of section 9) then Not(G) is true. For instance, an attempt to prove

Not(At(Block,b,begin)) will call SILLY(at(Block,b,begin) which will succeed. As currently defined SILLY will be over enthusiastic so Not will sometimes succeed when it should not (see sections 9,10).

This aspect of the new logic is not yet implemented. The reason is that the whole question of negation is a tricky one which needs a major rethink.

12. Conclusion

We have examined the properties of functions and seen that while they are a cause of the combinatorial explosion they cannot be dispensed with. We have explored a way of bringing this aspect of the combinatorial explosion under control by allowing limited creation of new terms. We have also shown how the uniqueness property of functions can be used to prune search. The incorporation of these control techniques into the inference mechanism has given rise to a new inference mechanism, which we hope will be the first step in the design of a computational logic, especially designed to facilitate inference by computer.

Why do we call our logic "computational"? Because the incorporation of search information in the inference mechanism only makes sense if you have an inference mechanism. That is, if you are designing a procedure for making inferences rather than a (static) logical system. In a conventional logical system (e.g. predicate calculus, lambda calculus) there is no sense of compulsion or advice. The rules of inference merely give you a range of options to take. It is not in the spirit of such a system to specify which options are to be taken first, which taken as a last resort and which not taken at all. Our computational logic has been designed to give such specifications.

13. References

Ballentyne, M. and Bennett, W. 1973. "Graphing Methods for Topological Proofs", Math.Dept. Memo ATP7, Austin, Texas.

Bundy, A. 1973. "Doing Arithmetic with Diagrams", Procs. of IJCAI-73, pp130-138, ed. Nilsson, N., Stanford, California.

Bundy, A., Luger, G., Stone, M. and Welham, R. 1976. "MECHO: Year One", Procs. of 2nd AISB Conf., pp94-103, ed. Brady, M., Edinburgh.

Bundy, A. 1977. "Will it reach the top? Prediction in the Mechanics World", DAI Research Report No. 31.

Gelernter, H. 1963. "Realization of a Geometry Theorem Proving Machine", Computers and Thought, pp134-152, eds. Feigenbaum and Feldman, McGraw Hill.

Kowalski, R. and Hayes, P. 1971. "Lecture Notes on Automatic Theorem Proving", DCL Memo 40, Edinburgh.

Nevins, A. 1974. "Plane Geometry Theorem Proving using Forward Chaining", MIT AI Memo No. 303.

Fikes, R.E. and Nilsson, N.J. 1971. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence, Vol.2, Nos.3/4, pp189-208

Warren, D., 1977. "Implementing PROLOG - compiling predicate logic programs", Vol. 1 & 2, DAI Research Reports Nos. 39 & 40, Edinburgh.

Welham, R., 1976. "Geometry Problem Solving", DAI Research Report No. 14, Edinburgh.

Wos, L., Robinson, G. and Carson, D.F., 1965. "Automatic Generation of Proofs in the Language of Mathematics", IFIP Congress 65 Proceedings, Vol.II, pp35-326, Spartan Books, Washington D.C.

Acknowledgements

My thanks are due to: Jerry Schwarz for valuable comments on an earlier draft; Bob Kowalski, Gordon Plotkin and numerous others for many conversations over several years; the SRC for supporting the research and Peggy for keeping typing.