D-SCRIPT: A COMPUTATIONAL THEORY OF DESCRIPTIONS

Robert C. Moore
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract

This paper describes D-SCRIPT, a language for representing knowledge in artificial Intelligence programs. D-SCRIPT contains a powerful formalism for descriptions, which permits the representation of statements that are problematical for other systems. Particular attention is paid to problems of opaque contexts, time contexts, and knowledge about knowledge. The design of a deductive system for this language is also considered.

1. Int roduction

1.1 Ways of Representing; Knowledge

Methods advocated for representing knowledge in artificiall intelligence programs have Included logical statements (McCarthy, Sandewall), semantic networks (Quillian, Schank), and procedures (Hewitt, Sussman and McDermott), All these approaches shpre one fundamental concept, the notion of predication. That Is, the basic data structure In each system Is some representation of a predicate applied to objects. In this respect, the various systems *are* more or less equivalent. But this basic idea must be extended to handle problems of quantification and knowledge about knowledge. Mere the systems do differ. We will argue, though, that these differences result from the descriptive apparatus used in the particular systems being compared, rather than from an inherent advantage of, say, procedures over declaratives *or* vice versa.

Advocates of PLANNER (e.g. Winograd, p. 2153 have argued that the predicate calculus cannot represent how a piece of knowledge should be used. But this is true only of the <u>first-order</u> predicate calculus. In a higher-order or non-ordered declarative language, statements could be made which would tell a theorem prover how other statements are to be used. PLANNER, on the other hand, has no way of directly stating an existential quantification, but this does not mean that procedural languages are necessarily Incapable of handling that problem.

Our belief, then. Is that the type of system used to represent knowledge Is unimportant, so long as It has sufficient expressive power. This paper presents an attempt at such a system, the language D-SCRIPT. As the name implies, the most Interesting feature of D-SCRIPT is Its powerful formalism for descriptions, which enables it to represent statements that are problematical in other systems. No position will be taken as to what kind of language D-SCRIPT is. Since It is intended to answer questions by making deductions from a data base, it can be thought of as a theorem prover. Since it operates by comparing expressions like the data-base languages of PLANNER and CONNIVER, It can be thought of as a pattern-matching language. And since It Is Turing universal and. In fact, includes the lambda calculus. It can be thought of as a programming language.

1.2 Problems In Representing Knowledge

Before presenting the details of D-SCRIPT, we will try to give some Idea of the type of problem It Is designed to solve. A classic problem Is that of representing opaque contexts. An opaque context is one which does not allow substitution of referentially equivalent expressions or does not allow existential quantification. For example the verb "want" creates an opaque context:

(1.1) John wants to marry the prettiest girl.

This sentence is ambiguous. it can mean either:

(1.2) John wants to merry a specific girl who also happens to bp the prettiest.

or:

(1.3) John wants to marry whoever is the prettiest girl, although he may not know who that is.

Under the first interpretation we can substitute any phrase which refers to the same person for "the prettiest girl". That is, if the prettiest girl is named "Sally Sunshine", from (1.2) we can infer:

U.I*) John wants to marry a specific girl who also happens to be named Sally Sunshine.

We cannot make the corresponding inference from (1.3). It will not be true that:

(1.5) John wants to *marry* whoever is named Sally Sunshine, although he may not know who that is.

Because of this property, (1.2) is called the transparent reading of (1.1) and (1.3) is called the opaque reading. It Is almost always the case that sentences having an opaque reading are ambiguous with the other reading being transparent.

To illustrate blocking of existential quantification, consider:

(1.6) John wants to marry a blonde.

Again the sentence Is ambiguous, meaning either:

(1.7) John wants to marry a specific girl, who also happens to be a blonde.

or:

(1.8) John has no particular girl In mind, but he wants whoever he does marry to be a blonde.

We can existentially quantify over the first reading but not the second. We can Infer:

(1.9) There exists someone whom John wants to marry.

from (1.7), but not from (I.8).

Another problem is the occurrence of descriptive phrases In sentences involvlng time reference. In the sentence:

(1.10) The President has been married since 1945.

the phrase "the President" refers to an Individual. In the sentence:

(1.11) The President has lived in the White Mouse since 1800.

"the President" refers to each President In turn.

Another type of sentence where the reference of a phrase depends on time Is Illustrated by:

(1.12) John met the President In 1960.

This sentence is ambiguous, but unlike (1.11), each Interpretation refers to only one person. The ambiguity Is whether "the President" refers to the President at the time (1.12) is asserted, or the President in 19C0.

representing knowledge about knowledge raises some interesting issues. For Instance, in:

(1.13) John knows Bill's phone numher.

how is John's knowledge to be represented? In John's mind It might be something like:

(1.14) (PHONE-NUM BILL 987-651(3)

So, (1.13) might be:

(1,15) (KNOWS JOHN (PHOMF-NUM BILL 987-6543))

The trouble with (1.15) is that It includes too much Information. Not only does It say what (1.13) says, it also says what the number La- The difficulty is to refer to a piece of Information without stating it.

For all these types of sentences, D-SCRIPT provides representations which allow the correct deductions to be made. Further, it provides separate representations for each meaning of the ambiguous sentences, and these representations are related In a way that explains the ambiguity.

2. The D-SCRIPT Language

2.1 P-SCRIPT Expressions

D-SCRIPT contains the following types of expressions:

1. constants
2. variables
3. forms
4. lists

A constant Is any alpha-numeric (I.e. only letters or numbers) character string (e.g. "F00", "BL0CK5"). A variable Is any alpha-numeric character string prefixed by "?" (e.g. "?X"). A form Is any sequence of expressions enclosed In angle-brackets (e.g. "<X Y ?Z>"). A list Is any sequence of expressions enclosed In parentheses (e.g. "(F00 A <BAR B C>)").

D-SCRIPT observes the convention that all functions, predicates, and operators evaluate their arguments. The rules for evaluating expressions are largely adapted from LISP. In fact, D-SCRIPT variables and forms are treated just like LISP atoms and lists, respectively. Rather than introducing "QUOTE", however, we use constants and lists to represent pre-defined Items. To state our rules formally:

1. A constant evaluates to Itself.

2. A variable evaluates to the expression which It has been assigned.

3. The value of a form Is the result of applying Its first element to the values of Its remaining elements. This will not be defined In general, but only for those expressions which represent meaningful operations In D-SCRIPT. One such case Is that of lambda-expressions. A lambda-expression is represented in D-SCRIPT by a form containing the constant "LAMBDA", followed by a list of variables, followed by an expression (e.g. "<LAMBDA (?X ?Y) <TIMES ?X ?Y>>"). A form whose first element Is a lambda-expression is evaluated in the same way as a corresponding LISP expression. The result is the value of the body of the lambda-expression, with the values of the arguments assigned to the corresponding variab les. For Instance, assuming "+" has the usual meaning, "<<LAMBDA (?X) <+ 2 ?X>> 3>" has the same value as "<+ 2 3>", which Is "5". We will introduce other types of forms whose value is defined when we explain the representation of statements.

4. A list evaluates to a form with Identical structure, except that free variables are replaced by their values. If "?X" has previously been assigned the value "A", then "(LAMBDA (?Y) CFOO ?X ?Y)) will evaluate to "<LAMBDA C?Y) CFOO A ?Y)>".

It is worth noting that the way lambda-expressions and lists are defined makes it very easy to write functions which construct complex forms. For example, consider "<LAMBDA C?X) (F00 (BAR (GRITCH ?X)))>". The result of applying this to "Z" is "<F00 (BAR (GRITCII Z))>". A comparable LISP function would have to be built up with "COMS" 's to achieve this result.

2.2 Representing Knowledge in D-SCRIPT

The most basic statements are those which express simple predication. A statement of this kind Is represented In D-SCRIPT by a form whose first element Is a constant representing the predicate and whose other elements are constants representing the objects of the predicate. For example:

(2.1) The sun is a star.
C2.2) BlockA is on BlockB.

could be represented as:

(2.3) <STAR SUN>
(2.It) <0N BLOCKA BL0CKB>

A simple statement about a statement, such as:

(2.5) John believes the sun Is a star.

would be:

(2.5) <BEL1EVE JOHN (STAR SUN)>

The Important thing to notice about (2.6) Is that the embedded statement Is represented by a list. This Is because we need an expression whose <u>value</u> Is (2.3) to be consistent with the convention that predicates (In this case, "believe") evaluate their arguments.

To represent more complex statements/ two types of extensions are needed. The simpler of these is the addition of logical connectives. D-SCR1PT uses "OR", "AND", "NOT", and "IMPUFS" to stand for the obvious logical operations. As In (2.6) the embedded statements are expressed as lists. So:

(2.7) If the sun is a star, then BlockA is on BlockB.

would be represented by:

(2.8) < IMPUFS (STAR SUN) (ON BLOCKA BL0CKB)>

This notation reflects the fact that in D-SCP.tPT, logical connectives operate on the statements themselves rather than on their truth-values. "IMPLIES", then, is not computed as a Boolean function, but rather is computed by asserting that its first argument Is true, and attempting to prove its second argument.

The other extension required for complex statements, and the one that is most Important to our theory, is the use of descriptions. There are three types of descriptions in D-SCRIPT; existential descriptions, universal descriptions and definite descriptions. A description is a form whose first element is "SOME" (existential), "EVERY" (universal), or "THE" (definite); whose second element is a list containing a variable; and whose third element Is an expression whose value is a statement. Descriptions represent the corresponding types of natural language descriptive phrases:

(2.9) a block <S0ME (?X) (CLOCK ?X)>
every number <FVFRY (?Y) (HUM ?Y)>
the Table <TIIF (?X) (TABLE ?X)> ,

Some examples of sentences containing descriptive phrases and their representations are:

(2.10) The king Is fat.
<FAT <THE (?X) (KING ?XI>>

(2.11) John owns a dog.
<0WN JOHN <S0ME (?X) (DOG ?X)>>

(2.12) Every boy likes Santa Claus.
<LIKE <EVERY (?X) (BOY ?X)> SANTA>

Notice that when descriptions appear in statements, they are left as forms. This Is because, unlike embedded statements, we are talking about the objects to which the descriptions refer (i.e. their values) rather than the descriptions themselves.

The notation we have used so far Is not sufficient to express statements containing more than one occurrence of the same description. In the sentence:

(2.13) Every boy either loves Santa Claus or hates him.

the phrase "every boy" is the subject of both "loves" and "hates". We cannot use the following representation though:

(2.14) <OR (LOVE <EVERY <?X> (BOY 7X>> SANTA)
(HATE <FVERY (?X) (BOY ?X)> SANTA)>

because this means:

(2.15) Either every boy loves Santa Claus or every boy hates Santa Claus.

which, of course, is quite different. We can overcome this difficulty by using lambda-expressions. We will represent (2.13) by:

(2.16) <<LAMBDA (?X> (OR (LOVE ?X SANTA)
(HATE ?X SANTA))>
<FVFRY (?Y) CBOY ?Y)>>

This can be read as something like "the predicate X is true of every boy," where the predicate X is "loves Santa Claus or hates him."

Vie have a similar situation with respect to the scope of quantifiers. it Is not clear whether:

(2.17) <GRFATER <S0ME <?X) (NUM ?X)>
<EVERY (?Y) (NUM ?Y)>>

represents:

(2.1&) For every number there is some larger number.

or:

(2.18) There Is some number which is larger than every number.

We will have to arbitrarily choose a rule to disambiguate (2,17), but by using lambda-expressions we can avoid the difficulty. (2,18) can be represented by:

(2.20) <<LAMBDA (?X)
(CRFATER <S0ME (?Y) (MUM ?Y)> ?X)>
<FVERY (?Z) (HUM ?Z)>>

and (2.19) can be represented by:

(2.21) <<LAMBDA (?X)
(GREATER ?X <EVERY (?Y) (MUM ?Y)>)>
<S0ME (?Z) (NUM ?Z)>>

Analyzing these expressions In the same way as (2.1C) will show that they have the correct meanlng.

It should be apparent that existential and universal descriptions In D-SCRIPT serve exactly the same function as the quantifiers of the predicate calculus. in view of this. It may be asked why we have used a different notation, one reason Is that our notation makes it possible to write expressions v/hose structure more ctosely resembles the sentences they represent. Hopefully this makes them more Intelligible. The more important reason, though, is that having a single expression for a description makes It easier for an interpreter to manipulate It.

2.3 Formal Semantics of P-SCRIPT

The previous two sections outlined the syntax and informal semantics of D-SCRIPT. This section attempts to show how a program could be written that would Interpret D-SCRIPT statements In accord with their Intuitive meaning. The details of this will be somewhat sketchy. One reason for this Is that choosing proof strategies and using heuristic Information are complicated problems that we cannot claim to have solved. Secondly, creating a theorem prover is not our main goal. What we are trying to do is to show the sort of descriptive system necessary to represent the information contained in natural language statements. The purpose of this section is to establish that our notation for that system is "well-founded".

The program we have in mind would take a statement as its Input and determine from Its data base whether the statement is true. For statements which are simple predications/ the program looks for another statement in the data base which matches the first statement. The statement whose truth is being determined will be called the "test statement"; the statement in the data base to which it is being compared will be called the "tarret statement". To prove a complex statement, the program would break It down into its components and process them according to the semantics of the operators involved. Similarly, a complex target statenent must be broken down to its components for processing, but the rules are different. So, in explaining the semantics of complex expressions, analyses will be given for their use both In test statements and in target statements.

Two basic statements match If their corresponding elements match. In general, expressions whtch are not statements match whenever their values are Identical, A variable which has not been assigned a value matches any expression, and Is assigned that expression's value. These rules apply to both test statements and target statements. As an example, suppose "5" has been assigned to "?X", "?Y" Is unassigned, and "+" has Its usual meaning. Then "<FOD 5 ?Y>" will match "<F00 ?X <+ 3 4>>" and "7" will be assigned to "?Y".

We will not give a complete deductive procedure for logical connectives. It Is a well understood problem and Is not of primary Importance in the phenomena we wish to explain. But to suggest the kind of procedure we have In mind, consider "AND" and "IMPLIES". In handling these expressions the distinction between test statements and target statements comes through. To prove "<AND X *Y"* both X and *x* must be proved; but in matchin something against "<AMD X *Y>",* the match succeeds if either X or V matches. "<MPLIES X Y>" Is true If In a hypothetical state where X is asserted, Y can be proved. A test statement will match a target statement "<IMPLIES X Y>*" if* the test statement matches *X* and X can be proved. "OR" and "NOT" are somewhat more complicated but can be handled In much the same way.

The really Important part of our deductive procedure Is the treatment of descriptions. Definite descriptions are the simplest. "<THE <?X) <...?X...)>" evaluates to the constant which when assigned to "?X" makes "<...?X...>" true. If there is not such a constant or If there is more than one, the value of the description Is undefined. For example. If "LESS" means "arithmetically less than", then "<F00 3>" matches:

(2.22) <F00 <THE (?X) (AND (LESS ?X 4*)*
(LESS 2 ?X))>>

This rule for evaluating definite descriptions applies to both test statements and target statements.

For existential and universal descriptions, there is again a difference between test statements and target statements. In a test statement, an existential description matches anything that makes the body of the description true. That is, "<FO0 <SOME (?X) (BAR ?X)>>" matches "<F00 A>" If "<BAR ?X>" Is true when "?X" Is assigned "A". For the case of a target statement, the evaluation is more difficult. If we know that "Some bar is foo," we could simply give It a name and continue. But giving a name would imply that we know which bar is foo, which is not true. Instead we can create a name and say that if the new name were the name of the object that is asserted to exist, then anything which we can prove about the new name Is true of the object. We do this by creating a hypothetical state of the data base in which. if the new name is "G999", we assert "<BAR G999>". The target statement then becomes "<F00 G999>". Another way of putting this is that "<SOME (?X> (BAR ?X)>" evaluates to "G999", with the side effect of creating a hypothetical state of the data base In which "<BAR 3999>" is asserted. When the hypothesis is discharged, the new name becomes undefined, and we are not in danger of supposing that we know v/hat the name of the object is.

The treatment of universal descriptions Is the exact dual of that for existential descriptions. In a test statement, we know that whatever we can prove about an arbitrarily selected member of a class is true of every member of the class. So just as we did for existential target statements, we set up a hypothetJcal state, produce an arbitrary unique name, and assert that It is a member of the class. Analogously to what we said before, "<EVERY (?X) (FOO ?X)>" evaluates to, say, "G111l" with the side effect of creating a hypothetical state In which "<F00 GIII>" is asserted. Also in duality with existential descriptions. In a target statement a universal description matches anything which makes its body true. For example, "<F00 A>" matches "<F00 <EVERY (?X) (BAR ?X)>>" If "<BAR ?X>" Is true when "?X" is assigned "A".

Now we can see why lambda-expressions are Important for representing Information in 0-SCRIPT. Evaluating existential and universal descriptions sometimes has the side effect of changing the data base. Later we will Introduce other expressions which also do this. If we have other descriptions In the statement, we need to be able to control whether they are evaluated In the old data base or the new. By "lambda-fylng" a statement we can bring one or another description to the outside and force It to be evaluated first. In this way we can control the order In which expressions are evaluated. A detailed example of this will be given in section 3.5.

In this brief summary, we have given the

barest outlines of a deductive procedure. We have not discussed any of the complex Interactions among these logical operators. But hopefully we have laid a sufficient foundation to talk about the Issues that are the real point of this paper.

3. Solution to Representation Problems Using D-SCRIPT

3.1 Descriptions In Opaque Contex 6

In general, descriptive phrases In opaque contexts are subject to more than one Interpretation. Furthermore, at least one of the Interpretations seems not to behave according to normal rules of logical manipulation. Looking more closely, opaoue contexts primarily occur In the complement constructions of verbs like "want", "believe", "know", etc. These verbs all have the property of describing somebody's model of the world. When we say:

(3.1) John wants to marry Sally.

what we mean Is that in John's model of the world, the state:

(3.2) John is married to Sally.

Is considered desirable. The ambiguity of descriptive phrases arises from the question of whether the descriptive phrase Is to be evaluated In our model of the world or the model of the subject of the sentence. To Illustrate this, recall the sentence:

(3.3) John wants to marry the prettiest girl.

In D-SCR1PT, the opaque reading is represented by:

(3.4) <WANT JOHN (MARRY JOHN
<THE (?X) (PRETTIFST ?X)>)>

The reason that there are restrictions on substituting other expressions for "<THE (?X) (PRETTIEST ?X)>" is that the statement which actually contains this description, I.e.:

(3.5) <MARRY JOHN <THE (?X) (PRETTIEST ?X)>>

Is part of John's world model. If in our program we represent John's world model by a separate data base, then the expressions which may be substituted are those which are equivalent In that data base, not In the main data base which represents our world model.

To represent the transparent reading of (3.3), we must take the description outside the scope of John's model. We can do this with a lambda-expression:

(3.6) <<LAMBDA (?X)
(WANT JOHN (MARRY JOHN ?X))>
<THE (?Y) (PRFTTIFST ?Y)>>

This says that the statement we get by evaluating the description In our model and substituting that value for "?X" In:

(3.7) <MARRY JOHN ?X>

Is marked as a desirable state In Jolin's world model.

The analysis is analogous for existential descriptions. The two readings of:

(3.8) John wants to marry a blonde,

can be represented by:

(3.9) <WANT JOHN (MARRY JOHN
<S0ME (?X) (BLONDE ?X)>)>

for the opaque reading, and by:

(3.10) <<LAMBDA (?X)
(WANT JOHN (MARRY JOHN ?X))>
<S0ME (?Y> (BLONDE ?Y)>>

for the transparent reading. (3.9) means:

(3.11) John, wants there to be a blonde that he marries.

and (3.10) means:

(3.12) There Is a blonde that John wants to marry.

So the reason we can't make a "there Is..." paraphrase of (3.9) Is that rather than being an existential statement. It Is an assertion abqut an existential statement.

3.2 Descriptions in time contexts

In order to discuss the next set of examples, we need a way to represent time. The basic fact here Is that any predicate can be made to vary with time. Even those that we choose to consider eternal can be alleged to depend on time, e.g.:

(3.13) Two used to be greater than three.

To account for this in first-order logic, we would have to make time an explicit parameter of every predicate symbol. Instead, we will represent time by a coptext-STRUCTURED.c.tu.red data base (McDermott). By this we mean that the data base will be broken down Into a series of sub-data bases, or contexts, each of which represents the state of the world at some particular time. This can be efficiently Implemented, as It is in CONNIVER (Sussman and McDermott) by specifying each context by recording the differences between It and Its predecessor.

To use this kind of data base, we need a special predicate "T-A-T" which takes as Its parameters a statement and the name of a time context. "<T-A-T S t>" means statement s. Is True At Time T. The formal semantics of "T-A-T" are that it attempts to deduce S, in the time context named by t., We also need to be able to generate references to time contexts. For Instance, the phrase:

(3.14) when Washington was President

would be represented by the description:

(3.15) <THE (?T) (T-A-T (PRES WASHINGTON) ?T)>

Finally we need the one-place predicate "TIME" to make quantified statements about time. We would represent:

(3.16) Three Is always greater than two.

by:

C3.17) <T-A-T (HRFATF.R 3 2)
<FVFRY (?T) (TIT'F ?T)>>

Given this notation for time, we can solve the associated problems which wo raised earlier. As in the case of opaque contexts, the solution depends on whether a description Is evaluated In the context in which a statement Is made or the context which the statement Is about. Recalling the previous examples:

(3.18) The President has been married since 1945.

is represented by:

(3.19) <<LAMBDA (?X) (T-A-T (MARRIED ?X)
<EVERY (?T) (AFTER ?T 1945)>)>
<THE C?Y) (PRES ?Y)>>

In (3.19) the use of the lambda-expression puts the description "<TNIF (?Y) (PRES ?Y)>" outside the time construction, so it is evaluated in the context in which the statement is made. On the other hand:

(3.20) The President has lived In the White Mouse since 1800.

is represented by:

(3.21) <T-A-T
(LIVE-If <TME (?X) (PRES ?X)> W-H)
<EVERY (?T) (AFTER ?T 1800)>>

Here the description Is inside the time construction and is not evaluated until the time description has been instantiated. The analysis is the. same for:

(3.22) John met the President in 1960.

except that in this case the time reference is definite. One interpretation is given by:

C3.23) <T-A-T (MEET JOHN <T11E (?X) (PRES ?X)>
19G0>

and the other is given by:

(3.2d) <<LAHDDA (?X) (T-A-T (MEET JOHN ?X)
1960)>
<THE (?Y) (PRES ?Y)>

3.3 Knowledge about Knowledge

One of the questions we raised in the beginning was how to represent:

(3.25) John knows Dili's phone number.

If we knew the number we could represent (3.25) by:

(3.2G) <KNOW JOHN (PHONE-NUM BILL xxx)>

where xxx is the number. We do know one description of the number, namely "CiU's phone number". If we substitute this Into (3.26), however, we pet a trivial statement:

(3.27) <KN0W JOHN (PHOME-NUM BILL
<TIIE (?X) (PHONF-NUM BILL ?X)>)>

which means:

(3.28) John knows that Bill's phone number is Bill's phone number.

What we need to do Is to remove the occurrence of the description from John's world model "into our world model. Once again, we can do this with a lambda-expression:

(3.29) <<LAMBDA (?X)
(KNOW JOHN (PHONE-NUM BILL ?X))>
<T!IF (?X) (PHONE-NUM BILL ?X)>>

This says that If we were to evaluate the description "Bill's phone number" and stick the result in (3.26), we would correctly describe John's knowledge.

To see the difference between (3.27) and (3.29), suppose we know that Bill has a phone number, and we know that John knows that Bill has a phone number. These facts are represented by:

(3.30) <PH0ME-MUM BILL <S0ME (?X) (MUM ?X)>>

(3.31) <KN0W JOHN (PHONE-NUM BILL
<SOME (?X) (NUM ?X)>)>

Given this, we can prove (3.29) from itself. Notice that In D-SCRIPT this is non-trvial. Complex statements are never proved by simply looking to see if they are in the late base. Rather, they *arc* broken down to their basic components and these components are processed according to the semantics of the operators combining then. In the case of "KMOW" the semantics are to shift the proof to the data base of the person doing the knowing. So even to prove a statement from itself, the semantics really have to work.

tn trying to prove (3.29) the lambda-expression makes us first evaluate "(THE (?X) (PHONE-NUM DILL ?X)>". We do this by trying to find a match for "<PHONE,-NUM BILL ?X>". If we don't know Bill's phone number wc can't do this directly. (3.30), however, entitles us to create a hypothetical state in which some arbitrary constant, say "0777" is asserted to be Dill's number. So to prove (3.29), wc attempt to prove:

(3.32) <KN0W JOHN (PHONE-NUM BILL G777)>

with the hypothesis:

(3.33) <PI!0NE-NUM BILL G777>

To prove (3.32) from (3.29) we process (3.29) much the same as before. This time, however, We already have (5.33) in the data base; so "<THE (?X) (PHONE-N'UM BILL ?X)>" evaluates to "0777" directly. Our proof then reduces to proving (3.32) from itself, which reduces again to proving (3.33) from itself in the data base which represents John's world model. (3.33) is a basic statement, so it can be Inferred from Itself immediately, and the entire proof succeeds.

Now suppose instead that we were trying to prove (3.29) from (3.27). The proof would be the same down to the point where we generated the subgoal of proving (3.32). To prove this from (3.27), we have to prove (3.33) from:

```
(3.34) <PHONE-NUM BILL
            <THE (?X) (PHONE-NUM BILL ?X)>>
```

In the context of John's world model. But this time we cannot use (3.33) when we evaluate the description, because (3.33) is asserted only in our world model, and the evaluation is taking place in John's. What will happen is that (3.31) will be used to generate another arbitrary constant (e.g. "G888") in John's world model. We will then try to prove (3.33) from:

```
(3.35) <PHONE-NUM BILL G888>
```

Since "G777" does not match "G888", the proof fails.

4. Future work

In this paper we have presented a formal Language for the representation of knowl edge. We have shown how Information which is difficult to express in other formalisms can be expressed in our language. And we have suggested how a deductive program could be designed to answer questions in our language. Clearly, the next step in this research is to build that deductive program.

There arc several reasons why this would be a worthwhile project. For one, A.I. deductive systems seen to fell into two extreme categories. On the one hand, predicate-calculus theorem provers restrict themselves to first order languages. Procedural systems such as PLANNER, on the other hand, use pattern matching schemes which are general enough to process higher order statements, but they are So general that they say nothing about the meaning of those statements. Implementing D-SCRIPT would create a system somewhere In between - one that would embody systematic knowledge about some types of higher order statements.

Beyond this, the particular types of knowledge we have discussed seem to be especially important for A.I. There is still much work to be done, but if we can program a deductive system to treat "T-A-T" and "KNOW" in the way we have proposed, we will have taken a first step towards creatine programs which can think about thinking.

Bibilography

Hewitt, C, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models In a Robot," Report AI TR-25B, M.I.T. A.I. Laboratory, 1972.

McCarthy, J., "Programs with Common Sense," in Semantic Information Processing. Marvin MInsky, ed,, pp. 403-418. Cambridge, Mass.: N. I.T. Press, 1068.

McDermott, D, V., "Assimilation of flew Information by a Natural LanGuare-UnderstandInn System," unpublished S.N.. thesis, M.I.T., 1973.

Quillian, M. R., "Semantic Memory, " in Semantic Information Processing. on. 227-270.

Sandewall, F., "Formal Methods in the Design of Question-Answering Systems,[11] Artificial Intelligence. Vol. 2 (1971), pp. 129-145.

Schank, R, C. "A Conceptual Dependency Representation for a Computer-Oriented Semantics," Memo AI-83, Stanford A.I. Project, 1969.

Sussman, G. J. and D. V. McDermott, "From PLANNER to CONNIVER - A genetic approach," Proc. FJCC 41 (1972), pp. 1171-1179.

Winograd, T., "Procedures as a Representation for Data In a Program for Understandinp; Natural Language," Report AI TR-17, M.I.T. A.I. Laboratory, 1971.

CONVERSION OF PREDICATE-CALCULUS AXIOMS, VIEWED AS NON-DETERMINISTIC PROGRAMS, TO CORRESPONDING DETERMINISTIC PROGRAMS

Erik Sandewall
Computer Sciences Department
Uppsala university

Abstract: The paper considers the problem of converting axioms in predicate calculus to deterministic programs, which are to be used as "rules" by a GPS-type supervisor. It is shown that this can be done, but that the "objects" must then contain procedure closures or "FUNARG-expressions" which are later applied.

Keywords: deduction, theorem-proving, retrieval, non-deterministic, closure, FUNARC-expression.

Background- Retrieval of Implicit information in a semantic data base is a kind of deduction. One approach to doing such retrieval has been resolution-style theorem-proving; a later approach has been high-level programming languages such as Planner[1] and QA4[2], where non-deterministic programs and pattern-directed invocation of procedures are available. The use of uniform proof procedures for this purpose has been repeatedly criticized, e.g. In [3]. Users of the high-level languages have also been worried because their systems are very expensive to use[4,2] and because the non-determinism is difficult to control[4].

There is another approach, which has roots in A.1. research back to the General Problem Solver[5], where one has a supervisor which administrates a (relatively) fixed set of operators, and a working set of active objects. In each cycle, the supervisor picks an object and an operator (using any heuristic information that it may have), applies the operator to the object, and obtains back a number of new objects (none, one, or more) which are put into the working set. This process is continued until some goal is achieved (e.g., an object is a given target set appears in the working set).

This approach has certain advantages from an efficiency standpoint, The operators are fixed programs, which can be compiled or otherwise transformed all the way to machine code level. The non-determinism is concentrated to the supervisor. Still, there is room for pattern-directed invocation, by letting the supervisor classify objects into a number of classes, and associating a subset of the operators with each class. There is also the non-determinism Implied by the search.

The major disadvantage, of course, is that this scheme is more rigid. For example, since everything happens on one level, there is little room for recursion. If one operator calls a procedure, which calls another, which wants to be non-deterministic, then there is no trivial way to map that non-determinism back up to the "search level" of the supervisor, while retaining the environment of function calls, variable bindings, etc. that must be kept available in all branches.

An Interesting question is therefore: how harmful is this rigidity? Is it very awkward to "program around" the limitations of such a system, or is It easy?

In this paper, we try to answer that question by studying those operators which correspond to axioms in predicate calculus. We assume that we have a data base, which Is like a large number of ground unit clauses, plus a number of operators, which should correspond to the non-ground axioms. We show that there are certain problems in phrasing the latter as operators,,but that there is a systematic way to handle those problems. We conclude that the search supervisor approach should be considered as a serious candidate for the deductive system associated with a data base.

Basic Idea. For the reader who might not want to read the whole paper, we disclose that the idea is to permit the "objects" to contain procedure closures[6,7], also called FUNARG-expresslons, I.e. lambda-expressions together with an environment of bindings for its free variables. The lambda-expression is as fixed as the set of operators, and can therefore be compiled, etc, but the environment Is new for each object.

After thus having sketched the background and the general idea, let us go Into the details of the predicate-calculus environment.

Simplest case. Let us take a common-place axiom and convert it into a program-like operator. We choose the transitivity axiom,

R(x,y) A R(y,z)) R(x,z)

which goes Into a rule of the form

```
On a sub-question with the relation R, use
lambda(x,z) begin local y;
            determine y from R(x,y);
            return sub-question R(y,z)
            end
```

Here, "determine y from R(y,z)" calls for a look-up in the data base, and usually acts as a non-deterministic assignment to y. "Return sub-question" specifies the information which is given back to the supervisor, consisting of a relation (R) and an argument list. The latter is a list of the current values of x and yj It does not need to contain the names x and y, or their bindings to their current values. The supervisor will then look up all operators (lambda-expressions) which are associated with R, and apply them to the given argument list, of course at whatever time it chooses.

This rule describes what has to be done when any data base search routine continues search according to the transitivity property of the relations. It does not matter if the search is executed by a uniform theorem-prover, a Planner-type system, or by a hand-tailored proaram such as the LISP functions in the SIR system[8]. However, in a higher-level system, the system has to "interpret" the axioms or rules, I.e. find out at run-time what is to be done. A resolution theorem-prover is extreme In this respect. Our concern In this paper is to find out before execution (with information only about the axiom or rule, not about the actual sub-question) what operations will be necessary, so that we can write out the code for doing exactly that, tn programming systems terms, we want to compile the axioms, and do as many decisions as possible at complle-time.

If a resolution theorem-prover contains the above transitivity axiom, and the axiom

R(a,b)

and if it asked the "question" ^(b.c), it will generate the sub-question Ml(a,c). This step can be clearly Illustrated If the transitivity axiom Is rewritten as

$R(x,y) \wedge \sim R(x,z) \supset \sim R(y,z)$
If the same effect is to be obtained in a Planner system or a hand-tailored program, it must be programmed separately. In analogy to the rule above, we would write

On a sub-question with the relation $\bar{R}$, use
lambda (y,z) begin local x;
determine x from R(x,y)
return sub-question $\sim R(x,z)$
end

Thus one clause (In the resolution sense) usually corresponds to several rules like the lambda-expressions above. The number of rules that correspond to a clause Is finite. If some rules are omitted, then the resulting system is not In general complete, but inclusion of all rules is still not sufficient to insure completeness. We shall not be concerned about this.

Going back to the first rule above, the reader should imagine that the supervisor contains one queue of sub-questions for each relation symbol, and that every sub-question contains an argument list. Every relation symbol is associated with a set of operators, written as lambda-expressions like the one above, which can be applied to the objects that queue for that relation symbol. The operator above returns a sub-question, and tells what object - argument list it should contain, and which relation it should attend. The operators can be thought about as "demons", clustered in groups with a common point of interest, which is named by the relation symbol.

List of problems. This organization raises a number of questions. One problem is how one should integrate heuristic information into the system. We shall not go into that question here. Another question is how the local non-determintsm in the rule is to be handled. The answer is simple: we map the linear (i.e. loop-free), non-deterministic program into a looping, deterministic program. Each branch-point starts a new loop inside the loops of the previous branch-points. All loops end at the end of the rule. This is quite straight-forward.

If the PC (predicate calculus) axioms contain function symbols (not merely relations), we obtain "unification", or in programming language terms: pattern-matching and pattern-reconstruction. Then the convers ion to remove the local non-determinism involves some additional problems, which however will be the topic of a later extension of this paper. Suffice it to say that every PC function should be associated with one construction procedure and one or more matching procedures, and that the compiled version of the axiom must contain a call to one of these procedures. It can be determined at "compilation time" which procedure shall be called. The matching procedure for "plus" may for example match "1*" against "plus(x,l)" and assign to "x" the value 3.

Let us turn instead to the question of how open questions are handled. ("Closed questions" are questions which can be answered with a truth-value, I.e. Yes/no questions; "open questions" are questions which have an Individual, or n-tuple of individuals as possible answer.) We decide immediately that "closed questions with the relation R" shall be one class of object and interest-point for operators, and "open questions with the relation R and an asked-for second argument, R(x,?)" shall be another class of objects, treated with another set of operators. We shall provisionally denote it as R2(X). For example, the same transitivity axiom for R also calls for the following operator:

On a sub-question with R_2, use
lambda (x) begin local y;
determine y from R(x,y);
return sub-question Ra(y) end

Examples of non-trivial cases. Consider the PC axiom

$P(x,y) \wedge Q(x,y) \supset R(x,y)$

This should be represented by the following rule:

On a sub-question with R_2, use
lambda (x) begin
return sub-question $P_2(x)$, but check that any answer to that sub-question satisfies
lambda (y) Q(x,y)
before accepting it
end

Here the rule returns to the supervisor a relation symbol, an argument list, and a remainder procedure which is to be used later. In this case, the remainder procedure is lambda (y) Q(x,y). Notice also that the current binding of the variable x must be available to that procedure, when it is later used. The variable x is a transfer variable in the sense of reference[9]. In other words, the remainder procedure is a procedure closure as defined under "Basic idea" above and x must be bound in its environment part.

If we have PC functions in the axiom, a similar situation may arise.Consider

$P(x,y) \wedge Q(y,z) \supset R(x,f(x,z))$

which would go into

On a sub-question with R_2, use
lambda (x) begin local y,z;
determine y from P(x,y);
return sub-question $Q_2(y)$, and for every answer to the sub-question, apply
lambda (z) f(x,z)
and return the result
end

Here we again return a sub-question which contains a remainder procedure with a transfer variable (x).

So in both of these examples there was an unexpected complication: a need for objects which "contain" references to procedures. Because of the increased complexity, the mapping from PC axiom to corresponding rule is far from trivial in such examples. We shall now specify how it can be done. The method will be developed through "refinement", i.e. we first describe the general idea and then modify it until it becomes sufficiently precise.

New formulation of operators. Let us first re-write the operators without reference to what sub-question is being returned. For the three axioms that we have already used as examples, we obtain:

Axiom 1 (transitivity of R)

On a sub-question with R_2, use
lambda (x) begin local y,z;
determine y from R(x,y);
determine z from R(y,z);
return answer z
end

Axiom 2 $P(x,y) \wedge Q(x,y) \supset R(x,y)$

On a sub-question with R_2, use
lambda (x) begin local y;
determine y from P(x,y);
determine that Q(x,y) [is in the data base];
return answer y
end

Axiom 3 $P(x,y) \wedge Q(y,z) \supset R(x,f(x,z))$

On a sub-question with R_2, use
lambda (x) begin local y,z;
determine y from P(x,y);

```
    determine z from Q(y,z).;
    return answer f(x,z)
    end
```

Each of these operators contains a main block, where each statement except the last one makes an access to the data base, for either a closed or an open question, (Every such statement corresponds to a literal in the original axiom). We have tacitly assumed that thosere-references should be "immediate", i.e. only use facts that are explicitly in the data base. However, it Is also possible to let such intermediate statements make their own search. If we maintain the idea that the operators should be deterministic programs, and all search should be managed by the supervisor, then the search In the Intermediate statement must be brought to an end before the execution of the operator can continue. It follows that in an intermediate statement we can only make a search which is "short" compared to the main search done by the supervisor.

Is it possible to use the latest formulation of the operator as it is7 All search would then be done In the intermediate statements (both "look up y" and "look up z" in the transitivity axiom, etc.) and the operator can return a final answer, rather than a sub-question for further search. This is correct, but clearly the supervisor is not used at all in this case.

However, given the last formulation of the operators, we can come back to the previous formulation by pick' Ing out one intermediate statement and decide that that is where the main search shall be done. In the first axiom, the main search Is most naturally done for "determine z". In the second axiom, our previous formulation does the main search for "determine y", although In principle It would also be possible to determine y In the shallow search of an intermediate statement, and then ask the supervisor to do main search in order to prove Q(x,y) for the selected y. In the third axiom, our previous formulation does main search to determine z, although It would also be possible to do main search for y, and to determine z and f(x,z) in the remainder procedure.

Conclusion from the discussion. We conclude that the general method to convert a pred'cate-calculus axiom to an operator should be:

(1) Assign a suitable order to the literals to the left of the implication sign. ("Suitable" will not be discussed in this paper). Change each literal l into the phrase
"determine v_1, v_2, ... v_j from l"
where the v_i are variables which occur in l but not in previous literals, or (if $j = 0$)
"determine that l"

(2) Add a final statement, such as "return success" (for closed questions) or "return answer y" (for open questions). Also enclose the block by a lambda-expression. The information for this is taken from the literal to the right of the implication sign, in the obvious way as exemplified for the above axioms.

(3) Decide which of the statements in the operator shall be handled by the extensive, top-level search which is managed by the supervisor. This Is called a controlled statement. Let the statements in the operator be

H, s2...........S_{k-1}, S_k,S_{k+r} -.. S_n

where s. Is the controlled statement.

{*)) Construct a new operator where the statements are

s_1, s_2, ... s_{k-1}, s_k*

where s_k* is the following statement:
return the sub-question s_k, with the provision that any answer to this sub-question shall be further processed by the following remainder procedure:

lambda (v_1, v_2, ... v_j) begin s_{k+1}, ... s_n end

where the v_i are the variables mentioned in step (1) which occur in s_k.

The same examples again. Let us check this method on the three axioms that we have used above. In all cases, we give rules which are to be used on an open sub-question with R_2:

Axiom 1 (transitivity of R)

```
lambda (x) begin local y;
           determine y from R(x,y);
           return sub-question
              determine z from R(y,z), with the re-
              mainder procedure
              lambda (z) return answer z
           end
```

The phrase "determine z from R(y,z)" can be more concisely expressed as $R_2(y)$. We use that in the next two examples:

Axiom 2 $P(x,y) \wedge Q(x,y) \supset R(x,y)$

```
lambda (x) begin
           return sub-question P2(x), with the re-
           mainder procedure
              lambda (y) begin
                          determine that Q(x,y);
                          return answer y
                          end
           end
```

Axiom 3 $P(x,y) \wedge Q(y,z) \supset R(x,f(x,z))$

```
lambda (x) begin local y;
           determine y from P(x,y);
           return the sub-question Q2(y), with the
           remainder procedure
              lambda (z) begin
                          return answer f(x,z)
                          end
           end.
```

We see that this third formulation is equivalent to the first formulation of the rules, although it contains more strict formulations. In axiom 2, the statement "determine that Q(x,y)" will "fail" if the relation can not be retrieved or proved in the data base, and then control will never be passed on to the next statement, where the answer y is returned to the supervisor. The formulations above use both the primitives "return sub-question" and "return answer" with the obvious meaning.

We notice also that the formulations are still locally non-deterministic, and that they must undergo the trivial transformation to a deterministic program with loops. We write this out for the first example; the others are analoguous:

```
lambda (x) begin local y;
           for every y in set of answers to R2(x) do
              begin
              return sub-question R2(y) with remain-
              der procedure
                 lambda (z) return answer z
                 end
           end
```

Multiple controlled statements. It is easily seen that the above rule in four steps can be generalized to the cases where there are several controlled statements, and top-level search is performed for each of them. For example, in axiom 2 we might wish to make extensive search both in order to determine y from P(x,y), and in order to prove Q(x,y). We must then have two nested remainder procedures. The resulting operator should have the form:

```
On a sub-question with R2, use
lambda (x) begin
           return sub-question P2(x), with remain-
           der procedure
           lambda (y) begin
                      return sub-question Q(x,y)
                      /a closed sub-question/ with
                      the remainder prodedure
                      lambda () return answer y
                      end
           end
```

We realize that "every answer" to a closed sub-question must be affirmative, i.e. as soon as it has proved Q(x,y), the above operator returns y.

Chains of sub-questions. The operators as formulated above return sub-questions consisting of a relation symbol, an argument list, and a remainder function, but they only accept the first two items. This means that the supervisor is responsible for administrating the remainder procedures. However, in a programming system where procedures are permitted as arguments (to other procedures), the responsibility can easily be taken by the operators and the programming system. We shall now describe how this can be done.

In closed and open questions, we add one more argument 9, which is the remainder procedure. The resulting argument lists (x,y,g) for R, (x,g) for R_2, etc., are the objects which our supervisor shall handle.

We then modify the examples so that g is introduced as an argument and applied to the returned answer. Thus the definite version of the rule for axiom 3 ia:

```
On a sub-question with R2, use
lambda (x,g) begin local y;
             determine y from P(x.y);
             return sub-question
                Q2(y, function(lambda (z) g(f(x,z))))
             end
```

The other rules are modified similarly. We notice that the sub-questions that this rule returns, contain two transfer variables: x and g. The bindings of these must be saved in the closure, and retained until the remainder procedure is used.

Let g' be the second argument of 02 in one particular use of the above operator. Clearly g[1] contains a reference to g, which itself pfesumably is a procedure closure, which was set up by a previous sub-question. As one sub-question generates another, a chain of closures is generated, where each one refers to its predecessor. When finally an answer is found to the last sub-question, the last procedure closure is applied in a return-answer statement; it calls its predecessor by using a procedure variable, as seen in the example, the predecessor calls its predecessor, and so on up the chain. In the original (top-level) qjestion, q is given as "return aeswer".

Discussion of applicability of the method. This procedure works In all cases where the non-determinlstic interrupt points (where another, parallel branch is permitted to attract attention) can be brought to the top-level block of the "operators", and not be hidden deeper down in recursion, in principle, the trick is that the control stack (the stack of function calls) is only one element deep at the interrupt points (containing the call from the supervisor to the operator), and then the control stack information, plus the information of how far we have gotten, can be put in one additional transfer variable. With this method, we have no control stack environment, but merely a variable-binding environment at the interrupt points, and this is exactly what FUNARG (or procedure closures) can handle.

We believe that this method is sufficiently powerful to handle e.g. all cases which may occur when PC axioms are mapped into rules, and probab 1y also a broader application.

A questionable feature of this method is that one must in principle decide at "compile-time" which retrievals are to be done by "big" search, and which are to be done by "short" intermediate statement (• non-controlled statement) search. In some applications this is OK, since some relations *are* only stored explicitly or almost explicitly; In others it may not be acceptable.

Requirements on the programming language. If the conversion from PC axiom to operator is to be done automatically, then the selected programming language must of course be able to generate and manipulate programs in the same language. LISP Is then an obvious choice. However, during the execution of the search, our requirement is instead that we must be able to create a procedure closure, and send it around as data. Some simulation languages, notably Simula 67[10] have this facility, as well as POP-2[11] and ECL[12]. LISP1.5 systems (a-list systems) provide it through the FUNARG feature. Later *LISP* systems (LISP 1.6, *original* BBN-LISP) *do not* provide it[7]. A method for providing FUNARG in BBN-LISP'type systems without undue loss of efficiency has been proposed In[9].

It has been suggested that the notion of a "remainder procedure", as used in this paper, is rather closely connected with the notion of "continuation", which has recently proved helpful in discussing the denotational semantics of programming languages[13].

Implementation. The author has participated in the development of a program, called PCDB (Predicate Calculus Data Base), which Is organized according to the search supervisor principle. This program was described in reference [14], and contains a compiler which accepts PC axioms and generates corresponding LISP programs. It also contains a simple supervisor, elaborate data base handling facilities, etc. which are needed. The present (1972) version of PCDB lets the supervisor administrate the remainder procedures in an ad hoc and not completely general way. A new compiler is being written, which will administrate them with FUNARG expressions as indicated in this paper. We hope to have it working at the time of the conference.

Acknowledgements. The following people in Uppsala have helped with the PCDB work: Lennart Drugge, Anders Haraldson, Rene" Reboh.

Sponsor: This research was supported by IBM Svenska AB.

References

1. C Hewitt
   Description and theoretical analysis (using schemata) of PLANNER, a language for proving theorems and manipulating models In a robot
   Ph.D. thesis, Dept. of mathematics, MIT, Cambridge, Mass. (1972)

2. J.F. Rullfson et al.
   QA4: a procedural basis for intuitive reasoning
   AI Center, Stanford Research Institute (1972)

3. D.B. Anderson and P.J. Hayes
   The logician's folly
   in the (European) AISB Bulletin, British Computer Society, 1972

4. G.J. Sussman
   Why conniving is better than planning
   MIT AI laboratory, 1972

5. A. Newell et al.
   Report on a general problem-solving program
   Proc. IFIP Congress 1959, p. 256

6. P.J. Landin
   The mechanical evaluation of expressions
   Computer Journal, Vol. 6 (1964), pp. 308-320

7. J. Moses
   The Function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem
   ACM SIGSAM bulletin No. 15 (1970)

9. B. Raphe1
   SIR: a computer program for semantic Information retrieval
   in Minsky, ed.: Semantic information processing
   MIT press, 1968

9. E. Sandewall
   A proposed solution to the FUNARG problem
   ACM SIGSAM bulletin No. 17 (1971)

10. Ole-Johan Dahl et at.
   Common Base Language
   Norwegian Computing Center, Oslo, 1970

1. R.M. Burstall et al.
   Programming in POP-2
   Edlnburhg Univ. Press, 1971

2. B. Wegbreit et al.
   ECL Programmer's Manual
   Harvard University, Cambridge, Mass. 1972

3. J. Reynolds
   Definitional interpreters for higher order programming languages
   Proceedings of an ACM Conference, Boston, Mass.,1972

4. E. Sandewall
   A programming tool for management of a predicate-calculus-oriented data base
   in Proceedings of the second International joint conference on Artificial intelligence, British Computer Society, London, 1971

A Universal Modular ACTOR Formalism for Artificial Intelligence

Carl Hewitt
Peter Bishop
Richard Steiger

Abstract

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired 1n a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, *or* semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

"Programs should not only work,
but they should appear to work as well."
PDP-1X Dogma

The PLANNER project is continuing research in natural and effective means for embedding knowledge in procedures. In the course of this work we have succeeded in unifying the formalism around one fundamental concept: the ACTOR. Intuitively, an ACTOR is an active agent which plays a role on cue according to a script" we" use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. Data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, demons, processes, contexts, and data bases can all be shown to be special cases of actors. All of the above are objects with certain useful modes of behavior. Our formalism shows how all of the modes of behavior can be defined in terms of one kind of behavior: sending messages to actors. An actor is always invoked uniformly in exactly the same way regardless of whether 1t behaves as a recursive function, data structure, or process.

"It is vain to multiply Entities beyond need."
William of Occam
"Monotheism is the Answer."

The unification and simplification of the formalisms for the procedural embedding of knowledge has a great many benefits for us:

FOUNDATIONS: The concept puts procedural semantics [the theory of how things operate] on a firmer basis. It will now be possible to do cleaner theoretical studies of the relation between procedural semantics and set-theoretic semantics such as model theories of the quantificational calculus and the lambda calculus.

LOGICAL CALCULAE: A procedural semantics is developed for the quantificational calculus. The logical constants FOR-ALL, THERE-EXISTS, AND, OR, NOT, and IMPLIES are defined as actors.

KNOWLEDGE BASED PROGRAMMING is programming in an environment which has a substantial knowledge base in the application area for which the programs are intended. The actor formalism aids knowledge based programming in the following ways: PROCEDURAL EMBEDDING of KNOWLEDGE, TRACING BEHAVIORAL DEPENDENCIES, and SUBSTANTIATING that ACTORS SATISFY their INTENTIONS.

INTENTIONS: Furthermore the confirmation of properties of procedures is made easier and more uniform. Every actor has an INTENTION which checks that the prerequisites and the context of the actor being sent the message are satisfied. The intention is the CONTRACT that the actor has with the outside world. How an actor fullfIlls its contract is its own business. By a SIMPLE BUG we mean an actor which does not satisfy its intention. We would like to eliminate simpTedebugging of actors by the META-EVALUATION of actors to show that they satisfy their Intentions. Suppose that there is an external audience of actors E which satisfy the intentions of the actors to which they send messages. Intuitively, the principle of ACTOR INDUCTION states that the intentions of all actions caused by E are in turn satisfied provided that the following condition holds:

```
If for each actor A
     the intention of A is satisfied =>
     that the intentions of all actors sent messages by A are satisfied.
```

Computational induction [Manna], structural induction [Burstall], and Peano induction are all special cases of ACTOR induction. Actor based intentions have the following advantages: The intention is decoupled from the actors it describes. Intentions of concurrent actions are more easily disentangled. We can more elegantly write intentions for dialogues between actors. The intentions are written 1n the same formalism as the procedures they describe. Thus for example intentions can have intentions. Because protection is an intrinsic property of actors, we hope to be able to deal with protection issues in the same straight forward manner as more conventional intentions. Intentions of data structures are handled by the same machinery as for all other actors.

COMPARATIVE SCHEMATOLOGY: The theory of comparative power of control structures is

extended and unified. The following hierarchy of control structures can be explicated by incrementally Increasing the power of the message sending primitive:

iterative---recursive---backtrack---+determinate- --universal

EDUCATION: The model is sufficiently natural and simple that it can be made the conceptual basis of the model of computation for students. In particular it can be used as the conceptual model for a generalization of Seymour Papert's "little man" model of LOGO. The model becomes a cooperating society of "little men" each of whom can address others with whom it is acquainted and politely request that some task be performed.

LEARNING and MODULARITY: Actors also enable us to teach computers more easily because they make it possible to incrementally add knowledge to procedures without having to rewrite all the knowledge which the computer already possesses. Incremental extensions can be incorporated and interfaced in a natural flexible manner. Protocol abstraction [Hewitt 1969; Hart, Nilsson, and Fixes 1972] can be generalized to actors so that procedures with an arbitrary control structure can be abstracted.

EXTENDABILITY: The model provides for only one extension mechanism: creating new actors. However, this mechanism is sufficient to obtain any semantic extension that might be desired.

PRIVACF and PROTECTION: Actors enable us to define effective and efficient protection schemes. Ordinary protection falls out as an efficient intrinsic property of actors. The protection is based on the concept of "use". Actors can be freely passed out since they will work only for actors which have the authority to use them. Mutually suspicious "memoryless" subsystems are easily and efficiently implemented. ACTORS are at least as powerful a protection mechanism as domains [Schroeder, Needham, etc.], access control lists [MULTICS], objects [Wulf 1972], and capabilities [Dennis, Plummer, Lampson]. Because actors are locally computationally universal and cannot be coerced there is reason to believe that they are a universal protection mechanism in the sense that all other protection mechanisms can be efficiently defined using actors. The most important issues in privacy and protection that remain unsolved are those involving intent and trust. We are currently considering ways in which our model can be further developed to address these problems.

SYNCHRONIZATION: It provides at least as powerful a synchronization mechanism as the multiple semaphore P operation with no busy waiting and guaranteed first in first out discipline on each resource. Synchronization actors are easier to use and substantiate than semaphores since they are directly tied to the control-data flow.

SIMULTANEOUS GOALS: The synchronization problem is actually a special case of the simultaneous goal problem. Each resource which is seized is the achievement and maintenance of one of a number of simultaneous goals. Recently Sussman has extended the previous theory of goal protection by making the protection guardians into a list of predicates which must be re-evaluated every time anything changes. We have generalized protection in our model by endowing each actor with a scheduler. We thus retain the advantages of local intentional semantics. A scheduler actor allows us to program EXCUSES for violation in case of need and to allow NEGOTIATION and re-negotiation between the actor which seeks to seize another and its scheduler. Richard Waldinger has pointed out that the task of sorting three numbers is a very elegant simple example illustrating the utility of incorporating these kinds of excuses for violating protection.

RESOURCE ALLOCATION: Each actor has a banker who can keep track of the resources used by the actors that are financed by the banker.

STRUCTURING: The actor point of view raises some interesting questions concerning the structure of programming.

STRUCTURED PROGRAMS: We maintain that actor communication is well-structured. Having no goto, interrupt, semphore, etc. constructs, they do not violate "the letter of the law." Some readers will probably feel that some actors exhibit "undisciplined" control flow. These distinctions can be formalized through the mathematical discipline of comparative schematology [Patterson and Hewitt].

STRUCTURED PROGRAMMING: Some authors have advocated top down programming. We find that our own programming style can be more accurately described as "middle out". We typically start with specifications for a large task which we would like to program. We refine these specifications attempting to create a program as rapidly as possible. This initial attempt to meet the specifications has the effect of causing us to change the specifications in two ways:

1: More specifications [features which we originally did not realize are important] are added to the definition of the task.

2: The specifications are generalized and combined to produce a task that is easier to implement and more suited to our real needs.

IMPLEMENTATION: Actors provide a very flexible implementation language. In fact we are carrying out the implementation entirely in the formalism itself. By so doing we obtain an implementation that is efficient and has an effective model of itself. The efficiency is gained by not having to incur the interpretive overhead of embedding the implementation in some other formalism. The model enables the formalism to answer questions about itself and to draw conclusions as to the impact of proposed changes in the Implementation.

ARCHITECTURE: Actors can be made the basis of the architecture of a computer which means that all the benefits listed above can be enforced and made efficient. Programs written for the machine are guaranteed to be syntactically properly nested. The basic unit of execution on an actor machine is sending a message in much the same way that the basic

unit of execution on present day machines is an Instruction. On a current generation machine in order to do an addition an add Instruction must be executed; so on an actor machine a hardware actor must be sent the operands to be added. There are no goto, semaphore, interrupt, etc. instructions on an ACTOR machine. An ACTOR machine can be built using the current hardware technology that is competitive with current generation machines.

"Now! Now!" cried the Queen. "Faster! Faster!"
Lewis Carroll

Current developments in hardware technology are making it economically attractive to run many physical processors in parallel. This leads to a "swarm of bees" style of programming. The actor formalism provides a coherent method for organizing and controlling all these processors. One way to build an ACTOR machine is to put each actor on a chip and build a decoding network so that each actor chip can address all the others. In certain applications parallel processing can greatly speed up the processing. For example with sufficient parallelism, garbage collection can be done 1n a time which is proportional to the logarithm of the storage collected instead of a time proportional to the amount of storage collected which is the best that a serial processor can do. Also the architecture looks very promising for parallel processing In the lower levels of computer audio and visual processing.

"All the world's a stage,
And all the men and women merely actors.
They have their exits and their entrances;
And one man in his time plays many parts."

"If it waddles like a duck, quacks like a duck, and otherwise behaves like a duck; then you can't tell that it isn't a duck."

Adding and Reorganizing Knowledge

Our aim is to build a firm procedural foundation for problem solving. The foundation attempts to be a matrix in which real world problem solving knowledge can be efficiently and naturally embedded. We envisage knowledge being embedded in a set of knowledge boxes with interfaces between the boxes. In constructing models we need the ability to embed more knowledge in the model without having to totally rewrite it. Certain kinds of additions can be easily encompassed by declarative formalisms such as the quantificational calculus by simply adding more axioms. Imperative formalisms such as actors do not automatically extend so easily. However, we are implementing mechanisms that allow a great deal of flexibility in adding new procedural knowledge. The mechanisms attempt to provide the following abilities;

PROCEDURAL EMBEDDING:. They provide the means by which knowledge can easily and naturally be embedded in processes so that it will be used as intended.

CONSERVATIVE EXTENSION: They enable new knowledge boxes to be added and interfaced between knowledge "Foxes.

MODULAR CONNECTIVITY: They make it possible to reorganize the interfaces between knowledge boxes.

MODULAR EQUIVALENCE: They guarantee that any box can be replaced by one which satisfies the previous interfaces.

Actors must provide interfaces so that the binding of interfaces between boxes can be controlled by knowledge of the domain of the problem. The right kind of interface promotes modularity because the procedures on the other side of the interface are not affected so long as the conventions of the interface are not changed. These interfaces aid in debugging since traps and checkpoints are conveniently placed there. More generally, formal conditions can be stated for the interfaces and confirmed once and for all.

Unification

We claim that there is a common Intellectual core to the following (now somewhat isolated) fields that can be characterized and investigated: digital circuit designers, data base designers, computer architecture designers, programming language designers, computer system architects.

"Our primary thesis is that there can and must exist a single language for software engineering which is usable at all stages of design from the initial conception through to the final stage in which the last bit 1s solidly 1n place on some hardware computing system."
Doug Ross

The time has come for the unification and integration of the facilities provided by the above designers into an intellectually coherent manageable whole. Current systems which separate the following intellectual capabilities with arbitrary boundaries are now obsolete.

"Know thyself".

We intend that our actors should have a useful working knowledge of themselves. That is, they should be able to answer reasonable questions about themselves and be able to trace the implications of proposed changes in their intentions. It might seem that having the implementation understand itself is a rather incestuous artificial intelligence domain but we believe that it is a good one for several reasons. The implementation of actors on a conventional computer Is a relatively large complex useful program which is not a toy. The implementation must adapt itself to a relatively unfavorable environment. Creating a model of itself should aid in showing how to create useful models of other large knowledge based programs since the implementation addresses a large number of difficult semantic issues. We have a number of experts on the domain that are very interested 1n formalizing and extending their knowledge. These experts are good programmers and have the time, motivations, and ability to

embed their knowledge and intentions in the formalism.

"The road to hell is paved with good intentions."

Once the experts put in some of their intentions they find that they have to put in a great deal more to convince the auditor of the consistency of their intentions and procedures. In this way we hope to make explicit all the behavioral assumptions that our implementation 1s relying upon. The domain is closed 1n the "sense""that the questions that can reasonably be asked do not lead to a vast body of other knowledge which would have to be formalized as well. The domain is limited in that 1t is possible to start with a small superficial model of actors and build up Incrementally. Any advance is immediately useful in aiding and motivating future advances. There 1s no hidden knowledge as the formalism is being entirely implemented in itself. The task is not complicated by unnecessary bad software engineering practices such as the use of gotos, interrupts, or semaphores.

Intrinsic Computation

We are approaching the problem from a behavioral [procedural] as opposed to an axiomatic approach. Our view is that objects are defined by their actors rather than by axiomatizing the properties of the operations that can be performed on them.

"Ask not what you can do to some actor;
but what the actor can [will?] do for you."

Alan Kay has called this the INTRINSIC as opposed to the EXTRINSIC approach to defining objects. Our model follows the following two fundamental principles of organizing behavior:

Control flow and data flow are inseparable.

Computation should be done intrinsically instead of extrinsically i.e. "Every actor should act for himself or delegate the responsibility [pass the buck] to an actor who will."

Although the fundamental principles are very general they have definite concrete consequences. For example they rule out the goto construct on the grounds that it does not allow a message to be passed to the place where control is going. Thus it violates the inseparability of control and data flow. Also the goto defines a semantic object [the code following the tag] which is not properly syntactically delimited thus possibly leading to programs which are not properly syntactically nested. Similarly the classical interrupt mechanism of present day machines violates the principle of intrinsic computation since it wrenches control away from whatever instruction is running when the interrupt strikes.

Hierarchies

The model provides for the following orthogonal hierarchies:

SCHEDULING: Every actor has a scheduler which determines when the actor actually acts after it 1s sent a message. The scheduler handles problems of synchronization. Another job of the scheduler [Rulifson] is to try to cause actors to act in an order such that their intentions will be satisfied.

INTENTIONS: Every actor has an intention which makes certain that the prerequisites and context of the actor being sent the message are satisfied. Intentions provide a certain amount of redundancy in the specifications of what is supposed to happen.

MONITORING: Every actor can have monitors which look over each message sent to the actor.

BINDING: Every actor can have a procedure for looking up the values of names that occur within 1t.

RESOURCE MANAGEMENT: Every actor has a banker which monitors the use of space and time.

Note that every actor had all of the above abilities and that each is done via an actor!

"A slow sort of country!" said the Queen. "Now here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

Lewis Carroll

The previous sentence may worry the reader a bit as she [he] might envisage an infinite chain of actions [such as banking] to be necessary in order to get anything done. We short circuit this by only requiring that it appear that each of the above activities 1s done each time an actor is sent a message.

"There's no use trying," she said: "one can't believe impossible things."

"I daresay you haven't had much practice," said the Queen. "When I was your age, I always did it for half-an-hour a day. Why, sometimes I've believed as many as six impossible things before breakfast."

Lewis Carroll

Each of the activities is locally defined and executed at the point of invocation. This allows the maximum possible degree of parallelism. Our model contrasts strongly with extrinsic quantIficatlonal calculus models which are forced into global noneffective statements 1n order to characterize the semantics.

"Global state considered harmful."

We consider language definition techniques [such as those used with the Vienna Definition Language] that require the semantics be defined in terms of the global computational state to be harmful. Formal penalties [such as the frame problem and the definition of simultaneity] must be paid even if the definition only effectively modifies local parts of the state. Local intrinsic models are better suited for our purposes.

Hardware

Procedural embedding should be carried to Its ultimate level: the architecture of the machine. Conceptually, the only objects in the machine are actors. In practice the machine recognizes certain actors as special cases to save speed and storage. We can easily reserve a portion of the name space for actors implemented in hardware.

Syntactic Sugar

"What's the good of Mercator's North Poles and Equators,
Tropics, Zones and Meridian Lines?"
So the Bellman would cry: and the crew would reply
"They are merely conventional signs!"
Lewis Carroll

Thus far 1n our discussion we have discussed the semantic issues intuitively but vaguely. We would now like to proceed with more precision. Unfortunately in order to do this it seems necessary to introduce a formal language. The precise nature of this language 1s completely unimportant so long as it 1s capable of expressing the semantic meanings we wish to convey. For some years we have been constructing a series of languages to express our evolving understanding of the above semantic issues. The latest of these is called PLANNER-73.

Meta-syntactic variables will be underlined. We shall assume that the reader 1s familiar with advanced pattern matching languages such as SN0B0L4, CONVERT, QA4, and PLANNER-71.

We shall use (%A M%) to indicate sending the message M to the actor A. We shall use [s1 s2 ... sn] to denote the finite sequence s1, s2_, ... sn. ft sequence s is an actor where (%s_ i%) is element i of the sequence s. For example (%[a c b] 2%) is c. We will use ( ) to delimit the simultaneous synchronous transmission of more than one message so that (Al A2...An) will be defined to be (%A1 [A2 ... An]%). The expression [%a1 a2 ... an%] (read as ""a] then a2 ... finally send back an") willl be evaluated by evaluating al, a2, ... and an in sequence and then sending back ["returning"] the value of an as the message.

Identifiers can be created by the prefix operator =. For example if the pattern = x 1s matched with y, then a new identifier is created and bound to v.

"But 'glory' doesn't mean 'a nice knock-down argument," Alice objected.
"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose 1t to mean—neither more nor less."
"The question is," said Alice, "whether you can make words mean so many different things."
"The question is," said Humpty Dumpty, "which is to be master-- that's all."
Lewis Carroll

Humpty Dumpty propounds two criteria on the rules for names:

Each actor has complete control over the names he uses.

All other actors must respect the meaning that an actor has chosen for a name.

We are encouraged to note that in addition to satisfying the criteria of Humpty Dumpty, our names also satisfy those subsequently proposed by Bill Wulf and Mary Shaw: The default is not necessarily to extend the scope of a name to any other actor. The right to access a name is by mutual agreement between the creating actor and each accessing actor. An access right to an actor and one of its acquan-tances is decoupled. It is possible to distinguish different types of access. The definition of a name, access to a name, and allocation of storage are decoupled. The use of the prefix = does not imply the allocation of any storage.

One of the simplest kinds of ACTORS is a cell. A cell with initial contents V can be created by evaluating (cell V_). Given a cell x, we can ask it to send back its contents by evaluating (contents xj which is an abbreviation for (x #contents). For example (contents(cell 3)) evaluates to 3. We can ask it to change its contents to v by evaluating (x-y_). For example if we let x be (cell 3) and evaluate (x--4), we will subsequently find" that (contents x) will evaluate to 4.

The pattern (by-reference P) matches object E_ 1f the pattern Pmatches (cell E) i.e. a "cell" [see below] which contains E. Thus matching the pattern (by-reference =*x)* against E 1s the same as binding x to (cell E) i.e. a new cell which contains the value of the expression E.We shall use => [read as "RECEIVE MESSAGE"] to mean an actor which is reminiscent of the actor LAMBDA in the lambda calculus. For example (=> x body) 1s like (LAMBDA x body) where x 1s an identifier. An expression (=> pattern body) is an abbreviation for (receive {[#message pattern]} body) where receive 1s a more general actor that is capable of binding elements of the action in addition to the message. Evaluating

(%(=> pattern body) the-message%), i.e. sending

(=> pattern body) the-message, will attempt to match the-message against pattern. If the-message is not of the form specified by pattern, then the actor is NOT APPLICABLE to the-message. If the-message matches pattern, then body 1s evaluated.

Evaluating (%(cases [f1 f2 ••• f_n]) arg%) will send fl_ the message arg and if it is not applicable then it will send f2 the message arg,..., and send fn the message aro

The following abbreviations will be used to improve readability:

(rules object clauses) for
((cases clauses)object)
(let object pattern-for-message body) for
(%(=> pattern-for-message body) objects)

```
                ;for example (let 3=x ( x+1)) is 4
(let-reception object pattern-for-reception body)
    (%(receive pattern-for-reception body) object%)
                ;let is a special case of let-reception
```

Sending Messages and Creating Actors

The world's a theatre, the earth a stage,
Which God and nature do with actors fill.
Thomas Heywood 1612

Conceptually at least a new actor is created every time a message is sent. Consider sending a message to a target T with message M and continuation C.

```
(send
    T
    (
        [#message the-body]
        [#continuation C]))
```

The transmission (%T M%) is an abbreviation for the above where C is defaulted to be the caller. If the target T is the following:

```
(receive
    (
        [#message M]
        [#continuation =the-continuation]}
    the-body)
```

then the-body is evaluated in an environment where the-message is bound to M and the-continuation is bound to C.

We define an EVENT to be a quadruple of the form [C T M N] where C is the continuation of the caller, T the target, and M the message thereby creating a new actor N. We define a HISTORY to be a strict partial order of events with the transitive closure of the partial ordering→ [read as *PRECEDES*] where

[c1 t1 m1 n1]→[c2 t2 m2 n2] if
{n1} intersect {c2 t2 m2} is nonvoid.

The above definition states that one action precedes another if any of the actors generated by the first event are used in the second event. The relation→can be thought of as the "arrow of time" which we require to be a strict partial order. Notice that we do not require a definition of global simultaneity; i.e. we do not require that two arbitrary events be related by →. We define BEHAVIOR of a history with respect to an AUDIENCE [a set of actors] E to be the subpartial ordering of the history consisting of those quadruples [C T M N] where C or T is an element of the audience E. The REPERTOIRE of a configuration of actors is the set of all behaviors of the configuration defines what the configuration does as opposed to how it does it. Two configurations of actors will be said to be EQUIVALENT if they have the same REPERTOIRE.

We can name an actor H with the name A in the body B by the notation (label {[A <= H]} B). More precisely, the behavior of the actor (label {[f <= (E f)]} B) is defined by the MINIMAL BEHAVIORAL FIXED POINT of (E f) i.e. the minimal repertoire F such that (E F) =F. In the case where F happens to define a function, it will be the case that the repertoire F is isomorphic the graph [set of ordered pairs] of the function defined by F and that the graph of F is also the least (lattice-theoretic) fixed point of Park and Scott.

Many Happy Returns.

Many actors who are executing in parallel can share the same continuation. They can all send a message ["return"] to the same continuation. This property of actors is heavily exploited in meta-evaluation and synchronization. An actor can be thought of as a kind of virtual processor that is never "busy" [in the sense that it cannot be sent a message].

The basic mechanism of sending a message preserves all relevant information and is entirely free of side effects. Hence it is most suitable for purposes of semantic definition of special cases of invocation and for debugging situations where more information needs to be preserved. However, if fast write-once optical memories are developed then it would be suitable to be implemented directly in hardware.

The following is an overview of what appears to be the behavior of the process of a running actor R sending a target T the message M specifying C as the continuation. If C is not explicitly specified by R then a representative of R must be constructed as the default.

1: Call the banker of R to approve the expenditure of resources by the caller.
2: The banker will probably eventually send a message to the scheduler of T.
3: The scheduler will probably eventually send a message to the monitors of T.
4: The monitors will probably eventually send a message to the intentions of T.
5: The intentions of T will probably eventually send the message M to the continuation of T.
6: The continuation of T will finally attempt to get some real work done.

There are several important things to know about the process of sending a message to an actor:

0: Conceptually at least, whenever a target is passed a message a new actor is constructed which is the target instantiated with a message. Wherever possible we reuse old actors where the reuse cannot be detected by the behavior of the system.

1: Sending messages between actors is a universal control primitivein the sense that control operations such as function calls, iteration, coroutine invocations, resource seizures, scheduling, sychroniztion, and continous evaluation of expressions are special cases.

2: Actors can conduct their dialogue directly with each other; they do not have to set up some intermediary such as ports [Krutat, Balzer, and Mitchell] or possibility lists [McDermott and Sussman] which act as pipes through which conversations must be conducted.

3: Sending a message to an actor is entirely free of side effects such as those in the

message mechanisms of the current SMALL TALK machine of Alan Kay and the port mechanism of Krutat and Balzer. Being free of side effects allows us a maximum of parallelism and allows an actor to be engaged in several conversations at the same time without becoming confused.

4: Sending a message to an actor makes no presupposition that the actor sent the message will ever send back a message to the continuation. The unidirectional nature of sending messages enables us to define iteration, monitors, coroutines, etc.straight forwardly,

5: The ACTOR model is nojt an [environment-pointer, instruction-pointer] model such as the CONTOUR model. A'continuation is a full blown actor [with all the rights and privileges]; it is not a program counter. There are no instructions [in the sense of present day machines] in our model. Instead of instructions, an actor machine has certain primitive actors built in hardware.

Logic

"It is behavior, not meaning, that counts."

We would like to show how actors represent formulas in the quantificational calculus and how the rules of natural deduction follow as special cases from the mechanism of extension worlds. We assume the existence of a function ANONYMOUS which generates a new name which has never before been encountered. Consider a formula of the form *(every* phi) which means that for *every* x we have that (phi x) is the case. The formula has two important uses: it can be asserted and it can be proved. We shall use an actor >=> [read as "ACCEPT REQUEST"] with the syntax

(>=> pattern-for-request body) for procedures to be invoked by pattern directed invocation by a command which matches pattern-for-request.

Our behavioral definitions are reminiscent of classical natural deduction except that we have four introduction and elimination rules [PROVE, DISPROVE, ASSERT, and DENY] to give us more flexibility in dealing with negation.

"Then Logic would take you by the throat, and force you to do it!"
Lewis Carroll

Data Bases

Data bases are actors which organize a set of actors for efficient retrieval. There are two primitive operations on data bases: PUT and GET. A new virgin data base can be created by evaluating (virgin). If we evaluate (w +■ (virgin)) then (contents w) will be a virgin world. We can put an actor (at John airport) in the world (contents w) by evaluating (put(at John airport) {[#world{contents w)]>). We could add further knowledge by evaluating

(put (at airport Boston) {[#world (contents w)]]) to record that the airport is at Boston.

(put {city Boston) {[#world (contents w)])) to record that Boston is a city.

If the constructor EXTENSION is passed a message then it will create a world which is an extension of its message. Eor example

```
(put
        [(on John (flight 34))
        (extension-world ■*- (contents w))])
```

will set extension-world to a new world in which we have supposed that John is on flight #34. The world (contents w) is unaffected by this operation. On the other hand the extension world is affected if we do (put [(hungry John) (contents w)]). Extension worlds are very good for modeling the following:

WORLD DIRECTED INVOCATION

The extension world machinery provides a very powerful invocation and parameter passing mechanism for procedures. The idea is that to invoke a procedure, first grow an extension world; then do a world directed invocation on the extension world. This mechanism generalizes the previous pattern directed invocation of PLANNER-67 several ways. Pattern directed invocation is a special case in which there is just one assertion in the wish world. World Directed Invocation represents a formalization of the useful problem solving technique known as "wishful thinking" which is invocation on the basis of a fragment of a micro-world. Terry Winograd uses restriction lists for the same purpose in his thesis version of the blocks world. Suppose that we want to find a bridge with a red top which is supported by its left-leg and its right-leg both of which are of the same color. In order to accomplish this we can call upon a genie with our wish as its message. The genie uses whatever domain dependent knowledge it has to try to realize the wish.

```
(realize
    (utopia
        (top left-leg right-leg color-of-legs}
            ;"the variables in the uptopia are listed above"
        {
            (color top red)
            (supported-by top left-leg)
            'supported-by top right-leg)
            ;ieft-of left-leg right-leg)
            [color left-leg color-of-legs)
            kcolor left-leg color-of-legs)}))
```

LOGICAL HYPOTHETICALS are logically possible alternatives to a world.

By the Normalization Theorem for intuitionistic logic our actor definition of the logical constant IMPLIES is sufficient to mechanize logical implication. The rules of natural deduction are a special case of our rules for extension worlds and our procedural definition of the logical connectives.

ALTERNATIVE WORLDS are physically possible alternatives to a world.

PERCEPTUAL VIEWPOINTS can be mechanized as extension worlds. For example suppose

rattle-trap is the name of a world which describes my car. Then (front rattle-trap) could be a world which describes my car from the front and (left rattle-trap) can be the description from the left side. We can also consider a future historian's view of the present by (vlew-from-1984 world-of-1972). MInsky [1973] considers these possibilities from a somewhat different point of view.

The following general principles hold for the use of extension worlds:

Each independent fact should be a separate assertion. For example to record that "the banana banl is under the table tabl" we would assert:

```
(banana banl)
 table tabl)
 under banl tabl)
```

instead of conglomerating [McDermott 1973] them Into one assertion:

```
(at
    (the banl (1s banl banana))
    (place
        (the tabl (is tabl table))
        under))
```

A person knowing a statement can be analyzed into the person believing the statement and the statement being true. So we might make the following definition of knowing:

```
[know <=
    (=> [= person = statement]
        (and
            (believes person statement)
            (true statement)))]
```

Thus the statement [Moore 1973] "John knows Bill's phone number" can be represented by the assertion:

```
(knows John (phone-number Bill pn0005))
```

where pn0005 is a new name and (phone-number Bill pn0005) is Intended to mean that the phone number of Bill 1s pn0005. The assertion can be expanded as follows:

```
(believes John (phone-number Bill pn0005))
(true (phone-number Bill pn0005))
```

However the expansion is optional since the two assertions are not independent of the original assertion.

> "Whatever Logic is good enough to tell me Is worth writing down," said the Tortoise. "So enter it in your book, please."
>
> Lewis Carroll

Each assertion should have justifications[derivations] which are also assertions and which therefore ...

Extraneous factors such as time and causality should not_ be conglomerated [McDermott 1973] into the extension world mechanism. Facts about time and causality should also be separate assertions. In this way we can deal more naturally and uniformly with questions involving more than one time. For example we can answer the question "How many times were there at most two cannibals in the boat while the missionaries and cannibals were crossing the river?" Also we can check the consistency of two different narratives of overlapping events such as might be generated by two people who attended the same party. Retreival of actors from data bases takes facts about time and causality into account 1n the retreival. Thus we still effectively avoid most of the frame problem of McCarthy. The ability to do this is enhanced by the way we define data bases as actors.

A CONTEXT mechanism was invented for QA4 to generalize the property list structure of LISP. Rulifson explained 1t by means of examples of Its use to mechanize identifiers. By use of the functions PUSH-CONTEXT and POPJONTEXT and an EPAM discrimination net [Feigenbaum and Simon] the context mechanism can be used to mechanize a version of tree-structured worlds, The tree-structured worlds of PLANNER-71 were Invented to get around the problem of having only one global data base not realizing that a context mechanism could be used to implement something like that. The tree-structured worlds were defined directly in terms of the hash-coding mechanism of PLANNER which had the advantage of decoupling them from the identifier structure of PLANNER. In addition by not conceiving an extension world analogue of POP_CONTEXT large gains in efficiency over the context mechanism are possible.

Worlds can ask the actors put In them to index themselves for rapid retreival.We also need to be able to retrieve actors from worlds. Simple retrieval can be done using patterns. For example

```
(locations +■ (get (at (?) (?)){[#world (contents w)]}))
```

will set locations to an actor which will retrieve all the actors stored in (contents w) which match the pattern (at (?) {?)). Now (next locations) will thus retrieve either (at airport Boston) or (at John airport). Actually* the above 1s an over simplification. We shall let $real1ty stand for the current world at any given point and $utopia stand for the world as we would like to see 1t. We do not want to have to explicity store every piece of knowledge which we have but would like to beable to derive conclusions from what is already known: We can distinguish several different classes of procedures for deriving conclusions.

"McCarthy 1s at the airport." (put (at McCarthy airport)) If a person 1s at the airport, then the person might take a plane from the airport,

```
[put-at <»
    (>«> (put (at = person airport))
        (put (might (take-plane-from person airport))))]
```

"McCarthy 1s not at the airport." (deny (at McCarthy airport)) If a person Is not at th airport then he can't take a plane from the airport.

"McCarthy is not at the airport." (deny (at McCarthy airport)) If a person is not at the airport then he can't take a plane from the airport.

```
[deny-at<=
    (>=> (deny (at =person airport))
        (put (can't (take—plane—from person airport ))))]
```

"It is not known whether McCarthy is at the airport," (erase (at McCarthy airport)) If it is not known whether a person is at the airport then erase whatever depends on previous knowledge that the person is at the airport,

```
[erase-at <=
    (>=> (erase (at -person airport))
        (find (depends—on =s (at person airport))
            (erase s)))]
```

"Get McCarthy to the airport." (achieve {(at McCarthy airport)}) To achieve a person at a place:

Find the present location of the person.
Show that it is walkable from the present location to the car.
Show that 1t is drivable from the car to the place,

```
[achieve-at <=
    (>=> (achieve [(at =person =place )])
        (achieve
            (find [(at person -present-location)]
                (show {(walkable present-location car)}
                    (show {(drivable car place)}))))))]
```

"Show that McCarthy is at the airport." (show {(at McCarthy airport)}) To show that a thing is at a place show that the thing is at some intermediate and the intermediate is at the place.

```
[show-at <=
    (>=> (show {(at =th1ng =place)})
        (show {(at thing 'intermediate)}
            (show {(at intermediate place)}))))]
```

The actor show-at is simply transitivity of at.

I! Anything Really Better Than Anything Else?

CONNIVER can easily be defined TrTTerms of PLAWR-73. We do this not because we believe that the procedures of CONNIVER are particularly well designed. Indeed we have given reasons above why these procedures are deficient. Rather we formally define these procedures to show how our model applies even to rather baroque control structures.

CONNIVER is essentially the conglomeration of the following ideas: Landin's non-hierarchical goto-71, the pattern directed construction, matching, retrieval, and invocation of PLANNER, Landin's streams, the context mechanism of QA4, and Balzer's and Krutar's ports.

In most cases, two procedures in CONNIVER do not talk directly to each other but instead are required to communicate through an intermediary which is called a possibilities list. The concept of a POSSIBILITIES LIST is the major original contribution of CONNIVER.

"What are these
So wild and withered in their attire,
That look not like the inhabitants
0' the earth,
and yet are on't?"

Macbeth: Act 1, Scene 111

Substitution, Reduction, and Meta-evaluation

"One program's constant is another program's variable."

Alan Perils

"Programming [or problem solving in general] is the judicious postponement of decisions and commitments!"

Edsger W. Dijkstra [1969]

"Programming languages should be designed to suppress what is constant and emphasize what is variable."

Alan Perlis

"Each constant will eventually be a variable!"

Corollary to Murphy's Law

We never do unsubstitution [or if you wish decompilation, unsimpllfication, or unevaluation]. We always save the higher level language and resubstltute. The metaphor of substitution followed by reduction gives us a macroscopic view of a large number of computational activities. We hope to show more precisely how all the following activities fit within the general scheme of substitution followed by reduction:

EVALUATION [Church, McCarthy, Lnadin] can be done by substituting the message into the code and reducing [execution].

DEDUCTION [Herbrand, Godel, Heyting. Prawltz, Robinson, Hewitt, Weyhrauch and Milner] can be done by procedural embedding. In this paper we have extended our previous work by defining the logical constants to be certain actors thus providing a procedural semantics for the quantlficational calculus along the lines indicated by natural deduction.

CONFIRMING the CONSISTENCY of ACTORS and their INTENTIONS [Naur, Floyd, Hewitt

1971, Waldlnger, Deutsch] can be done by substituting the code for the actors Into their intentions and then meta-evaluating the code.

AUTOMATIC ACTOR GENERATION. An important corollary of the Thesis of Procedural Embedding is that the Fundamental Technique of Artificial Intelligence is automatic programming and procedural knowledge base construction. It can be done by the following' "methods:

PARAMETERIZATION [Church, McCarthy, Landin, Mcintosh, Manna and Waldinger, Hewitt] of canned procedure templates.

COMPILATION [Lombardi, Elcock, Fikes, Daniels, Wulff, Reynolds, and Wegbreit] can be done by substituting the values of the free variables in the code and then reducing [optimizing]. For examples we can enhance the behavior of the lists which were behaviorally defined above to vectors which will run more efficiently on current generation machines.

ABSTRACT IMPOSSIBILITIES REMOVAL can be done by binding the alternatives with the code and deleting those which can never succeed, What we have in mind are situations such as having simultaneous subgoals (on a b) and (on b c) where we can show by meta-evaluation that the order given above can never succeed. Gerry Sussman has designed a program which attempts to abstract this fact from running on concrete examples. We believe that in this case and many others it can be abstractly derived by meta-evaluation.

EXAMPLE EXPANSION [Hart, Nilsson, and Fikes 1971; Sussman 1972; Hewitt 1971] can be done by binding the high level goal oriented language to an example problem and then reducing [executing and expanding to the paths executed] using world directed invocation [or some generalization] to create linkages between the variablized special cases.

PROTOCOL ABSTRACTION [Hewitt 1969, 1971] can be done by binding together the protocols, reducing the resulting protocol tree by identifying indistinguishable nodes.

ABSTRACT CASE GENERATION to distinguish the methods to achieve a goal can be done by determining the necessary pre-conditions for each method by reducing to a decision tree which distinguishes each method.

Acknowledgements

"Everything of importance has been said before by somebody who did not discover it."

Alfred North Whitehead

This research was sponsored by the MIT Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research. We would be *very* appreciative of any comments, criticisms, or suggestions that the reader might care to offer. Please address them to:

Carl Hewitt
Room 813
545 Technology Square
M.I.T. Artificial Intelligence Laboratory
Cambridge, Massachusetts 02139

The topics discussed in this paper have been under intense investigation by a large number of researchers for a decade. In this paper we have merely attempted to construct a coherent manageable formalism that embraces the ideas that are currently "in the air".

We would like to acknowledge the help of the following colleagues: Bill Gosper who knew the truth all along: "A data structure is nothing but a stupid programming language." Alan Kay whose FLEX and SMALL TALK machines have influenced our work. Alan emphasized the crucial importance of using intentional definitions of data structures and of passing messages to them. This paper explores the consequences of generalizing the message mechanism of SMALL TALK and SIMULA-67; the port mechanism of Krutar, Balzer, and Mitchell; and the previous CALL statement of PLANNER-71 to a universal communications mechanism. Alan has been extremely helpful in discussions both of overall philosophy and technical details. Nick Pippenger for his very beautiful ITERATE statememt and for helping us to find a fast economical decoding net for our ACTOR machine. John McCarthy for making the first circular definition of an effective problem solving formalism and for emphasizing the importance of the epistemological problem for artificial intelligence. Seymour Papert for his "little man" metaphor for computation. Allen Newell whose kernel approach to building software systems has here perhaps been carried to near its ultimate extreme along one dimension. David Marr whose penetrating questions led us to further discoveries. Rudy Krutar, Bob Balzer, and Jim Mitchell who introduced the notion of a PORT which we have generalized into an ACTOR. Robin Milner is tackling the problems of L-values and processes from the point of view of the lambda calculus. He has enphasized the practical as well as the theoretical implications of fixed point operators. Robin's puzzlement over the meaning of "equality" for processes led to our definition of behavior. Edsger Dijkstra for a pleasant afternoon discussion. Jim Mitchell has patiently explained the systems implementation language MPS. Jeff Rulifson, Bruce Anderson, Gregg Pfister, and Julian Davies showed us how to clean up and generalize certain aspects of PLANNER-71. Peter Landin and John Reynolds for emphasizing the importance of continuations for defining control structures. Warren Teitleman who cleaned up and generalized the means of integrating editors and debuggers in higher level languages. Peter Landin, Arthur Evans, and John Reynolds for emphasizing the importance of "functional" data structures. Danny Bobrow and Ben Wegbreit who originated an implementation method that cuts down on some of the overhead. We have simplified their scheme

by eliminating the reference counts and all of their primitives, c. A. R. Hoare 1s Independently Investigating "monitors" for data structures. Jack Dennis for sharing many of our same goals in his COMMON BASE LANGUAGE and for his emphasis on logical clarity of language definition and the importance of parallelism. Bill Wulff for our "." notation on the conventions of the values of cells and for being a strong advocate of exceptional cleanliness in language. Pitts Jarvis and Richard Greenblatt have given us valuable help and advice on systems aspects. Todd Matson, Brian Smith, Irene Grief, and Henry Baker are aiding us 1n the implementation. Chris Reeve» Bruce Daniels, Terry Winograd, Jerry Sussman, Gene Charniak, Gordon Benedict, Gary Peskin, and Drew McDermott for implementing previous generations of these ideas in addition to their own. J.C.R. Licklider for emphasizing the importance of mediating procedure calls. Butler Lampson for the notion of a banker and for the question which led to our criteria for separating an actor from its base. Richard Weyhrauch for pointing out that logicians are also considering the possibility of procedural semantics for logic. He is doing some *very* interesting research in the much abused field of "computational logic." Terry Winograd, Donald Eastlake, Bob Frankston, Jerry Sussman, Ira Goldstein, and others who made valuable suggestions at a seminar which we gave at M.I.T. John Shockley for helping us to eradicate an Infestation of bugs from this document. Greg Pfister, Bruce Daniels, Seymour Papert, Bruce Anderson, Andee Rubin, Allen Brown, Terry Winograd, Dave Waltz, Nick Horn, Ken Harrenstien, David Marr, Ellis Cohen, Ira Goldstein, Steve Zilles, Roger Hale, and Richard Howell made valuable comments and suggestions on previous versions of this paper.

Bibliography

Balzer, R.M., "Ports—A Method for Dynamic Interprogram Communication and Job Control"The Rand Corp., 1971.

Bishop, Peter, "Data Types for Programming Generality"M.S. June 1972. M.I.T.

Bobrow D., and Wegbreit Ben. "A Model and Stack Implementation of Multiple Environments." March 1973.

Davies, D.J.M. "POPLER: A POP-2PLANNER" MIP-89. School of A.I. University of Edinburgh.

Deutsch L.P. "An Interactive Program Verifier" Phd. University of California at Berkeley. June, 1973 Forthcoming.

Earley, Jay. "Toward an Understanding of Data Structures" Computer Science Department, University of California, Berkeley.

Elcock, E.W.; Foster, J.M.; Gray, P.M.D.; McGregor, H.H.; and Murray A.M. Abset, a Programming Language Based on Sets: Motivation and Examples. Machine Intelligence 6. Edinburgh, University Press.

Fisher. D.A. "Control Structures for Programming Languages" Phd. Carnegie. 1970

Gentzen G. "Collected Papers of Gerhard Gentzen".North Holland. 1969.

Greif I.G. "Induction in Proofs about Programs" Project MAC Technical Report 93. Feb. 1972.

Hewitt, C. and Patterson M. "Comparative Schematology" Record of Project MAC Conference on Concurrent Systems and Parallel Computation. June 2-5, 1970. Available from ACM.

Hewitt, C., Bishop P., and Steiger R. "The Democratic Ethos or 'How a Society of Noncoercable ACTORS can be Incorporated into a Structured System'" SIGPLAN-SIGOPS Interface Meeting, Savannah, Georgia. April, 1973.

Hewitt, C, and Greif,I. "Actor Induction and Meta-Evaluation"ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Boston, Mass- Oct, 1973. Forthcoming.

Hoare, C.A.R. "An Axiomatic Definition of the Programming Language PASCAL" Feb. 1972.

Kay, Alan C. Private Communication.

Krutar, R. "Conversational Systems Programming (or Program Plagiarism made Easy)" First USA-Japan Computer Conference. October 1972,

Lampson, B. "An Overview of CAL-TSS". Computer Center, University of California, Berkeley.

Liskov, B.H. "A Design Methodology for Reliable Software Systems" The Last FJCC. Dec.1972. Pt. 1, 191-199.

McDermott D.V. "Assimilation of New Information by a Natural Language-Understanding System" M.S. MIT. Forthcoming 1973.

McDermott, D.V. and Sussman G.J. "The Conniver Reference Manual" A.I. Memo no. 259. 1972.

Milner, R. Private Communication.

Minsky, Marvin. "Frame-Systems: A Theory for Representation of Knowledge" Forthcoming 1973.

Mitchell, J.G. "A Unified Sequential Control Structure Model" NIC 16816. Forthcoming.

Newell, A. "Some Problems of Basic Organization in Problem-Solving Programs." Self-Organizing Systems. 1962.

Papert S. and Solomon C. "NIM: A Game-Playing Program" A.I. Memo no. 254.

Reynolds, J.C. "Definitional Interpreters for Higher-Order Programming Languages" Proceedings of ACM National Convention 1972.

Rulifson Johns F., Derksen J.A., and Waldinger R.J. "QA4: A Procedural Calculus for Intuitive Reasoning" Phd. Stanford. November 1972.

Scott, D. "Data Types as Lattices" Notes. Amsterdam, June 1972.

Steiger, R. "Actors". M.S. 1973. Forthcoming.

Sussman, G.J. "Teaching of Procedures-Progress Report" Oct. 1972. A.I. Memo no. 270.

Waldinger R. Private Communication.

Wang A. and Dahl 0. "Coroutine Sequencing in a Block Structured Environment" BIT 11 425-449.

Weyhrauch, R. and Milner R. "Programming Semantics and Correctness in a Mechanized Logic." First USA-Japan Computer Conference. October 1972.

Winograd, T. "Procedures as a Representation for Data 1n a Computer Program for Understanding Natural Language" MAC TR-B4. February 1971.

W1rth, N. "How to Live without Interrupts" or some such. Vol. 1? No. 9, pp. 489-498.

Wulf W. and Shaw M. "Global Variable Considered Harmful" Carnegie-Mellon University. Pittsburgh, Pa. SIGPLAN Bulletin. 1973.

A MODEL FOR CONTROL STRUCTURES FOR ARTIFICIAL INTELLIGENCE PROGRAMMING LANGUAGES

by

Daniel G. Bobrow

Computer Science Division
Xerox Palo Alto Research Center
Palo Alto, California 94304

Ben Wegbreit

Harvard University
Center for Research in Computing Technology
Cambridge, Massachusetts 02138

Abstract

Newer programming languages for artificial intelligence extend the class of available control regimes beyond simple hierarchical control. In so doing, a key issue is using a model that clearly exhibits the relation between modules, processes, access environments, and control environments. This paper presents a model which is applicable to diverse languages and presents a set of control primitives which provide a concise basis on which one can define almost all known regimes of control.

1. Introduction

Newer programming languages! for artificial intelligence (e.g., PLANNER9^ CONNIVER,[18] BBN-LISP.ly QA4.1'1 extend the nature of control regimes available to the user. In this paper, we present an information structure model20 which deals with control and access contexts in a programming language; it is based on consideration of the form of run-time data structures which represent program control and variable bindings. The model is designed to help clarify some relationships of hierarchical function calls, backtracking, co-routines, and multiprocess structure. We present the model and its small set of primitive operations, then define several control regimes in terms of the primitives, and then consider extensions to handle cooperating sequential processes.

2. The Basic Environment Structure

In a language which has blocks and procedures, new nomenclature (named variables) can be introduced either by declarations in block heads or through named parameters to procedures. Since both define access environments, we call the body of a procedure or block a uniform access module. Upon entry to an access module, certain storage is allocated for those new named items which are defined at entry. We call this named allocated storage the basic frame of the module. In addition, certain additional storage for the module may be required for temporary intermediate results of computation; this additional allocated storage we call the frame extension. The total storage is called the total frame for the module, or usually just the module frame.

A" frame contains other information, in addition to named variables and temporaries. It is often useful to reference a frame by symbolic nomenclature. For this purpose, each frame has a framename (usually the procedure name). When a module is entered, its frame extension is initialized with two pointers (perhaps implicitly); one, called A LINK, is a linked access pointer to the frame(s) which contains the higher level free variable and parameter bindings accessible within this module. The other, called CLINK, is associated with control and is a generalized return which points to the calling frame. In Algol, these are called the static and dynamic links, respectively. In LISpH the two pointers usually reference the same frame, since bindings for variables free in a module are found by tracing up the call structure chain. (An exception is the use of functional arguments, and we illustrate that below.)

At the time of a call (entry to a lower module), the caller stores in his frame extension a continuation point for the computation. Since the continuation point is stored in the caller, the generalized return is simply a pointer to the last active frame.

The size of a basic frame is fixed on module entry. It is just large enough to store the parameters and associated information. However, during one function activation, the required size of the frame extension can vary widely (with a computable maximum), since the amount of temporary storage used by this module before calling different lower modules is quite variable. Therefore, the allocation of these two frame segments may sometimes (advantageously) be done separately and n on contiguously. This requires a link (BLINK) from the frame extension to the basic frame which contains the bindings.

When a frame is exited, either by a normal exit or by a non-local goto which skips the frame (e. g. , an error condition), it is often useful to perform clean-up action for the frame. Examples include: close files opened by the frame which are no longer needed, restore the state of more global structures which have been temporarily modified by the frame, etc. Terminal action for a frame is carried out by executing an exit function for the frame, passing it as argument the nominal value which the frame is returning as its result; the value returned by the exit function is the actual value of the frame. The variable values and the exit function are the only components of the frame which can be updated by the user; all the others are fixed at the time of frame allocation. Figure 1 summarizes the contents of the frame.

Figure 2a shows a sketch of an algorithm programmed in a block structure language such as Algol 60 with contourslO drawn around access modules. Bl has locals N and P, P has parameter N, and B3 locals Q and L. Figure 2b is a snapshot of the environment structure after the following sequence: Bl is entered; P is called (just above P1, the program continuation point after this outer call); B3 is entered; and F is called from within B3. For each access module there are two separate segments — one for the basic frame (denoted by the module name) and one for the frame extension (denoted by the module name*). Note that the sequence of access links (shown with dotted lines) goes directly from P to B1* and is different from the control chain of calls. However, each points higher

(earlier) on the stack.

A point to note about an access module is that it has no knowledge of any module below it. If an appropriate value (i.e., one whose type agrees with the stored return type) is provided, continuation in that access module can be achieved with only a pointer to the continued frame. No information stored outside this frame is necessary.

Figure 3 shows two examples in which more than one independent environment structure is maintained. In Figure 3a, two coroutines are shown which share common access and control environment A. Note that the frame extension of A has been copied so that returns from B and Q may go to different continuation points. This is a key point in the model; whenever a frame extension is required for conflicting purposes, a copy is made. Since frame A is used by two processes, if either coroutine were deleted, the basic frame for A should not be deleted. However, one frame extension A* could be deleted in that case, since frame extensions are never referenced directly by more than one process. Since the basic frame A is shared, either process can update the variable bindings in it; such changes are seen both by B and *Q,* In Figure 3b, coroutine Q is shown calling a function D with external access chain through B, but with control to return to Q.

3. Primitive Functions

In this model for access module activation, each frame is generally released upon exit of that module. Only if a frame is still referenced is it retained. All non-chained references to a frame (and to the environment structure it heads) are made through a special protected data type called an environment descriptor, abbreviated ed. The heads of all environment chains are referenced only from this space of descriptors. (The one exception is the implicit ed for the currently active process.) The primitive functions create an ed for a specified frame and update the contents of an ed; create a new frame with specified contents, and allow execution of a computation in that context; and access and update the exit function for a frame. Note that none of the primitives manipulate the links of existing frames; therefore, only well-formed frame chains exist (i.e., no ring structures).

1) environ(pos) — creates an environment descriptor for the frame specified by pos.

2) setenv(olded, pos) -- changes the contents of an existing environment descriptor olded to point to the frame specified by pos. As a side effect, it releases storage referenced only through previous contents of olded.

3) mkframe(epos,apos, epos,bpos,bcopflg) -- creates a new frame and returns an ed for that frame. The frame extension is copied from the frame specified by epos, and the ALINK and CLINK are specified by apos and epos, respectively. The BLINK points to the basic frame specified by bpos, or to a copy of the basic frame if bcopflg=TRUE. In use, arguments may be omitted; bcopflg is defaulted to FALSE; apos, bpos and epos are defaulted to the corresponding fields of the frame specified by epos. Thus mkframe(epos) creates a new frame extension identical to that specified by epos.

4) enveval(forrA,apos,cpos) — creates a new frame and initiates a computation with this environment structure. ALINK and CLINK point to frames specified by apos and epos, respectively; and form specifies the code to be executed, or the expression to be evaluated in this new environment. If apos or cpos are omitted, they are defaulted to the ALINK or CLINK of this invocation of enveval. Thus, enveval(form) is the usual call to an interpreter, and has the same effect as if the value of form had appeared in place of the simple call to enveval.

5) setexfn(pos,fn) — places a pointer to a user defined function in the exitfn field of the frame pos. If the system is using the exitfn, this will create a new function which is the composition of the user function (applied first) and the system function. On frame exit, the exitfn will be called with one argument, the value returned by the frame code; the value returned by fn will be the actual value returned to the frame specified by CLINK.

6) getexfn(pos) — gets the user set function stored in exitfn of frame pos. Returns NIL if none has been explicitly stored there.

7) framenm(pos) -- returns the framename of frame pos.

A frame specification (i.e. , pos, apos, bpos, epos, epos above} is one of the following:

1. An integer N:
   a. N=0 specifies the frame allocated on activation of the function environ, setenv, etc. In the case of environ, setenv and mkframe, the continuation point is set up so that a value returned to this frame (using enveval) is returned as a value of the original call to environ, setenv or mkframe.
   b. N>0 specifies the frame N links down the control link chain from the N=0 frame.
   c. N<0 specifies the frame INI links down the access link chain from the N=0 frame,

2. A list of two elements (F,N) where F is a framename and N is an integer. This gives the Nth frame with name F, where a positive (negative) value for N specifies the control (access) chain environment.

3. The distinguished constant NIL. As an access-link specification, NIL specifies that only global values are to be used free. A process which returns along a NIL control-link will halt. Doing a setenv(ed,NIL) releases frame storage formerly referenced only through ed, without tying up any new storage.

4. An ed (environment descriptor). When given an ed argument created by a prior call on environ, environ creates a new descriptor with the same contents as ed; setenv copies the contents of ed into olded.

5. A list "(ed)" consisting of exactly one ed. The contents of the listed ed are used identically to that of an unlisted ed. However, after this value is used in any of the functions, setenv(ed.NIL) is done, thus releasing the frame storage formerly referenced only through ed. This has been combined into an argument form rather than allowing the user to do a setenv explicitly because in the call to enveval the contents are needed, so it cannot be done before the call; it cannot be done explicitly after the enveval since control might never return to that point.

4. Non-Primitive Control Functions

To illustrate the use of these primitive control functions, we explain a number of control regimes which differ from the usual nested function call-return hierarchical structure, and define their control structure routines in terms of the primitives. We include stack jumps, function closure, and several multiprocessing disciplines. In programming examples, we use the syntax and semantics of a LISP-like system.

In an ordinary hierarchical control structure

system, if module F calls G, G calls H, and H calls J, it is impossible for J to return to F without going back through G and H. Consider some program in which a search is implemented as a series of such nested function calls. Suppose J discovered that the call to G was inappropriate and wanted to return to F with such a message. In a hierarchical control structure, H and G would both have to be prepared to pass such a message back. However, in general, the function J should not have to know how to force intermediaries; it should be able to pass control directly to the relevant module. Two functions may be defined to allow such jumpbacks. (These are implemented in BBN-LISP;19 experience has shown them to be quite useful.) The first function, retfrom(form,pos), evaluates form in the current context, and returns its value from the frame specified by pos to that frame's caller; in the above example, this returns a value to G's caller, i.e., P, The second function, retevaKform, pos), evaluates form in the context of the caller of pos and returns the "value of the form to that caller. These are easily defined in terms of enveval:

```
retfrom(form,pos) = enveval{form,2,pos)
retevalform, pos) = envevalform, pos, pos)
```

(The second argument to retfrom establishes that the current environment is to be used for the evaluation of form.)

As another example of the use of retfrom, consider an implementation of the LISP error protection mechanism. The programmer "wraps a form in errorset", i.e., errorset(form) which is defined as cons(eval(form),NIL). This "wrapping" indicates to the system the programmer's intent that any errors which arise in the evaluation of form are to be handled by the function containing the errorset. Since the value of errorset in the non-error case is always a list consisting of one element (the value of form), an error can be indicated by forcing errorset to return any non-list item. Hence, the system function error can be defined as retfrom(NIL,(ERRORSET 1)) where uppercase items are literal objects in LISP. This jumps back over all intermediary calls to return NIL as the value of the most recent occurrence of errorset in the hierarchical calling sequence.

In the following, we employ envapply which takes as arguments a function name and list of (already evaluated) arguments for that function. Envapply simply creates the appropriate form for enveval.

```
envapply(fn,args,aframe, cframe) =
    enveval(list(APPLY , list(QUOTE, fn),
            list(QUOTE, args)), aframe, cframe)
```

A central notion for control structures is a pairing of a function with an environment for its evaluation. Following LISP, we call such an object a funarg. Funargs are created by the procedure function, defined

```
function(fn)=list(FUNARG, fn, environ(2))
```

That is, in our implementation, a funarg is a list of three elements: the indicator FUNARG, a function, and an environment descriptor. (The argument to environ makes it reference the frame which called function.) A funarg list, being a globally valid data structure, can be passed as an argument, returned as a result, or assigned as the value of appropriately typed variables. When the language evaluator gets a form (fen arg1 arg2 ... argn) whose functional object fen is a funarg, i. e. , a list (FUNARG fn-name ed), it creates a list, args, of (the values of) argl, arg2, ... argn and does

```
envapply(second(fcn),args,third(fcn), 1)
```

The environment in this case is used exactly like the original LISP A-list. Moses 12 and Weizenbaum[25] have discussed the use of function for preserving binding contexts. Figure 4 illustrates the environment structure where a functional has been passed down: the function foo with variables X and L has been called; foo called mapcar(X,function(fie)) and fie has been entered. Note that along the access chain the first free L seen in fie is bound in foo, although there is a bound variable L in mapcar which occurs first in the control chain. Since frames are retained, a funarg can be returned to higher contexts and still work. (Burge[3] gives examples of the use of funargs passed up as values.)

In the above description, the environment pointer is used only to save the access environment. In fact, however, the pointer records the state of a process at the instant of some call, having both access and control environments. Hence, such an environment pointer serves as part of a process handle. It is convenient to additionally specify an action to take when the process is restarted and some information to be passed to that process from the one restarting it. The funarg can be reinterpreted to provide these features. The function component specifies the first module to be run in a restarted process, and the arguments (evaluated in the caller) provided to that function can be used to pass information. Hence, a funarg can be used as a complete process handle. It proves convenient for a running process to be able to reference its own process handle. To make this simple, we adopt the convention that the global variable curproc is kept updated to the current running process.

With this introduction, we now define the routines start and resume, which allow control to pass among a set of coordinated sequential processes, i.e. , coroutines, in which each maintains its own control and access environment (with perhaps some sharing). A coroutine system consists of n coroutines each of which has a funarg handle on those other coroutines to which it may transfer control. To initiate a process represented by the funarg fp, use start (we use brackets below to delimit comments):

```
start(fp.args) = curproc — fp;
      [ curproc is a global variable set to
       the current process funarg ] ;
      envapply(second(fp),args,third(fp),third(fp))
```

Once the variable curproc *is* initialized, and any coroutine started, resume will transfer control between n coroutines. The control point saved is just outside the resume, and the user specifies a function (backfn) to be called when control returns, i.e., the process is resumed. This function is destructively inserted in the funarg list. The args to this function are specified by the coroutine transferring back to this point.

```
resume(fnarg,args,backfn) =
    second( curproc) — backfn;
        [save the specified backfn for a subsequent
         resume back here]
    setenv(third(curproc), 2);
        [environment saved is the caller of resume]
    curproc — fnarg;
        [set up curproc for the coroutine to be
         activated]
    envapply(second(fnarg),args,third(fnarg),
            third (fnarg))
        [activate the specified coroutine by applying
         its backfn to args]
```

We call a funarg used in this way a process funarg. The state of a "process" is updated by destructively modifying a list to change its continuation function, and similarly directly modifying its environment descriptor in the list. A pseudo-multiprocessing capability can be added to the system using these process funargs if each process takes responsibility for requesting additional time for processing from a supervisor or by explicitly passing control as in CONNIVER,[18] A more automatic multiprocessing control regime using interrupts is discussed later.

Backtracking is a technique by which certain environments are saved before a function return, and later restored if needed. Control is restored in a strictly last saved, first restored order. As an example of its use, consider a function which returns one (selected) value from a set of computed values but can effectively return an alternative selection if the first selection was inadequate. That is, the current process can fail back to a previously specified failset point and then redo the computation with a new selection. A sequence of different selections can lead to a stack of failset points, and successive fails can restart at each in turn. Backtracking thus provides a way of doing a depth-first search of a tree with return to previous branch points.

We define fail and failset below. We use push(L,a) which adds a to the front of *L,* and pop(L) which removes one element and returns the first element of L. Failist is the stack of failset points. As defined below, fail can reverse certain changes when returning to the previous failset point by explicit direction at the point of failure. (To automatically undo certain side effects and binding changes, we could define "undoable" functions which add to failist forms whose evaluation will reset appropriate cells. Fail could then eval all forms through the next ed and then call enveval.)

```
failset{ ) = push(failist,environ(2))
      [2 means environment outside failset]
fail(message) = enveval(message,list(pop(failist)))
```

The function select defined below returns the first element of its argument set when first called; upon subsequent fails back to select, successive elements from set are returned. If set is exhausted, failure is propagated back. The code uses the fact that the binding environment saved by failset shares the variable fig with the instance of select which calls failset. The test of fig is reached in two ways: after a call on failset (in which *case fig* is false) *and* after a failure (in which case fig is true).

```
select(set,undolist) =
     progt  (fig)
s1:  if  null(set) then fail(undolist) [leave here and
                                      undo as specified]
     fig  — false;
     failsetOT
     [fig is true iff we have failed to this point; then
                               set has been popped]
     if fig then go(sl);
     fig — true;
     returnTpop(set));
     end
```

Floyd,"[7] Hewitt,9 and Golomb and Baumert[8] have discussed uses for backtracking in problem solving. Sussman[18] has discussed a number of problems with backtracking. In general, it proves to be too simple a form of switching between environments. Use of the multiple process feature described above provides much more flexibility.

5. Coordinated Sequential Processes and Parallel Processing

It should be noted that in the model above, control must be explicitly transferred from one active environment to another (by means of enveval or resume). We use the term, coordinated sequential process, to describe such a control regime. There are situations in which a problem statement is simplified by taking a quite different point of view - assuming parallel (cooperating sequential) processes which synchronize only when required (e. g. , by means of Dijkstra's[4] P and V operations). Using our coordinated sequential processes with interrupts, we can define such a control regime.

In our model of environment structures, there is a tree formed by the control links, a dendrarchy of frames. One terminal node is marked for activity by the current control bubble (the point where the language evaluator is operating). All other terminal nodes are referenced by environment descriptors or by an access link pointer of a frame in the tree. To extend the model to multiple parallel processes in a single processor system, k branches of the tree must be simultaneously marked active. Then the control bubble of the processor must be switched from one active node to another according to some scheduling algorithm.

To implement cooperating sequential processes in our model, it is simplest to think of adjoining to the set of processes a distinguished process, PS, which acts as a supervisor or monitor. This monitor schedules processes for service and maintains several privileged data structures (e.g. , queues for semaphores and active processes). (A related technique is used by Premier,[14])

The basic functions necessary to manipulate parallel processes allow process activation, stopping, continuing, synchronization and status querying. In a single processor coordinated sequential process model, these can all be defined by calls (through enveval) to the monitor PS. Specifications for these functions are;

1) process(form ,apos, cpos) -- this is similar to enveval except that it creates a new active process P' for the evaluation of form, and returns to the creating process a process descriptor (pd) which acts as a handle on P'.

In this model, the pd could be a pointer to a list which has been placed on a "runnable" queue in PS, and which is interpreted by PS when the scheduler in PS gives this process a time quantum. One element of the process descriptor gives the status of the process, e.g. , RUNNING or STOPPED. Process is defined using environ (to obtain an environment descriptor used as part of the pd) and enveval (to call PS),

2) stop(pd) — halts the execution of the process specified by pd — PS removes the process from runnable queue. The value returned is an ed of the current environment of pd.
3) continue(pd) -- returns pd to the runnable queues.
4) status(pd) — value is an indication of status of pd.
5) obtain(semaphore) — this Dijkstra P operator transfers control to PS (by enveval) which determines if *a resource* is available (i. e,, semaphore count positive). PS either hands control back to PI (with enveval) having decremented the the semaphore count, or enters P1 on that semaphore's queue in PS's environment and switches control to a runnable process.
6) release(semaphore) -- this *Dijkstra V operator* increments the semaphore count; if the count goes positive, one process is moved from the semaphore queue (if any exist) onto the runnable queue and the count is decremented. It then hands control back to the calling process.

We emphasize that these six functions can be defined in terms of the control primitives of section 3.

Scheduling of runnable processes could be done by having each process by agreement ask for a time resource, i.e., obtain(time), at appropriate intervals. In this scheduling model, control never leaves a process without its knowledge, and the monitor simply acts as a bookkeeping mechanism. Alternatively, ordinary time-sharing among processes on a time quantum basis could be implemented through a timer interrupt mechanism. Interrupts are treated as forced

calls to environ (to obtain an ed for the current state), and then an enveval to the monitor process. The only problem which must be handled by the system in forcing the call to environ is making sure the interrupted process is in a clean state; that is, one in which basic communication assumptions about states of pointers, queues, buffers, etc. are true (e.g., no pointers in machine registers which should be traced during garbage collection). This can be ensured if asynchronous hardware interrupts perform only minimal necessary operations, and set a software interrupt flag. Software checks made before procedure calls, returns and backward jumps within program will ensure that a timely response in a clean state will occur.

The ed of the interrupted process is sufficient to restart it, and can be saved on the runnable queue within a process descriptor. Because timer interrupts are asynchronous with other processing in such a simulated multiprocessor system, evaluation of forms in the dynamic environment of another running process cannot be done consistently; however, the ed obtained from stopping a process provides a consistent environment. Because of this interrupt asynchrony, in order to ensure system integrity, queue and semaphore management must be uninterruptible, e.g., at the highest priority level.

Obtaining a system of cooperating sequential processes as an extension of the primitives has a number of desirable attributes. Most important, perhaps, it allows the scheduler to be defined by the user. When parallel processes are used to realize a breadth-first search of an or-graph, there is a significant issue of how the competing processes are to be allotted time. Provision for a user supplied scheduler establishes a framework in which an intelligent allocation algorithm can be employed.

Once a multi-process supervisor is defined, a variety of additional control structures may be readily created. As an example, consider multiple parallel returns — the ability to return from a single activation of a module G several times with several (different) values. For G to return to its caller with value given by val and still continue to run, G simply calls process(val, 1,2). Then the current G and the new process proceed in parallel.

6. Conclusion

In providing linguistic facilities more complex than hierarchical control, a key problem is finding a model that clearly exhibits the relation between processes, access modules, and their environment. This paper has presented a model which is applicable to languages as diverse as LISP, APL and PL/I and can be used for the essential aspects of control and access in each. The control primitives provide a small basis on which one can define almost all known regimes of control.

Although not stressed in this paper, there is an implementation for the model which is perfectly general, yet for several subcases (e.g., simple recursion and backtracking) this implementation is as efficient as existing special techniques. The main ideas of the implementation are as follows (cf. [2] for details). The basic frame and frame extension are treated as potentially discontiguous segments. When a frame extension is to be used for running, it is copied to an open stack end if not there already, so that ordinary nested calls can use simple stack discipline for storage management. Reference counts are combined with a count propagation technique to ensure that only those frames are kept which are still in use.

Thus, the model provides both a linguistic framework for expressing control regimes, and a practical basis for an implementation. It is being incorporated intoBBN-LISP.19

7. Acknowledgments

This work was supported in part by the Advanced Research Projects Agency under Contracts DAHC 15-71-00088 and F19628-68-0-0379, and by the U.S. Air Force Electronics Systems Division under Contract F19628-71-C-0173. Daniel Bobrow was at Bolt Beranek and Newman, Cambridge, Massachusetts, when many of the ideas in this paper were first developed.

References

[l] Bobrow, D.G., "Requirements for Advanced Programming Systems for List Processing," CACM, Vol. 15, No. 6, June 1972.

[2l Bobrow, D.G. and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments," BBN Report No. 2334, Cambridge, Mass., March 1972, to appear in CACM.

[3] Burge, W.H. "Some Examples of the Use of Function Producing Functions," Second Symposium on Symbolic and Algebraic Manipulation, AC:M, 1971.

[41 Dijkstra, E.W. "Co-operating Sequential Processes," in Genuys (Ed.), Programming Languages, Academic Press, 1967.

[5] Dijkstra, E.W. "Recursive Programming," Numerische Mathematik 2 (l960), 312-318. Also in Programming Systems and Languages, S. Rosen (Ed.), McGraw-Hill, New York, 1967.

[6] Fenichel, R. "On Implementation of Label Variables, CACM, Vol. 14, No. 5 (May 1971), pp. 349-350.

[7] Floyd, R.W. "Non-deterministic Algorithms," J. ACM, 14 (October 1967), pp. 638-644.

[8t Golomb, S.W. and Baumert, L.D. "Backtrack Programming," J. ACM, 12 (October 1965), pp. 516-524.

[9! Hewitt, C. "PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot," *in* Artificial Intelligence, Washington, D.C., May 1969.

[10] Johnston, J.B. "The Contour Model of Block Structured Processes," in Tou and Wegner, Proc. Symposium on Data Structures in Programming Languages. SIGPLAN Notices, Vol. 6, No. 2, pp. 55-82.

[11] McCarthy, J., et al. Lisp 1. 5 Programmer's Manual, TheM.I.T. Press, Cambridge, Massachusetts (1962).

[12] Moses, J. "The Function of FUNCTION in LISP," SIGSAM Bulletin, No. 15, (July 1970), pp. 13-27.

[13j Prenner, C,, Spitzen, J. and Wegbreit, B. "An Implementation of Backtracking for Programming Languages," submitted for publication, ACM-72.

[14J Prenner, C. "Multi-path Control Structures for Programming Languages," Ph.D. Thesis, Harvard University, May 1972.

[151 Quam, L. LISP 1. 6 Reference Manual, Stanford AI Laboratory.

[16] Reynolds, J. "GEDANKEN - A Simple TypelesS Language Based on the Principle of Completeness and the Reference Concept," CACM, Vol. 13, No. 5 (May 1970), pp. 308-319.

[17] Rulifson, J. et al. "QA4- A Language for Writing Problem-Solving Programs," SRI Technical Note 48, November 1970.

[18] Sussman, G.J. "Why Conniving is Better than Planning," FJCC 1972, pp. 1171-1179.

[19] Teitelman, W., Bobrow, D. , Murphy, D., and Hartley, A. BBN-LISP Manual. BBN, July 1971.

[20] Tou, J, andWegner, P. (Eds.), SIGFLAN Notices — Proc. Symposium on Data Structures in Programming languages. Vol. 6, No. 2 (February 1971)

[2Il van Wijngaarden, A. (Ed.). Report on the Algorithmic Language ALGOL 68, MR 101, Mathematisch Centrum, Amsterdam (February 1969).

[22] Wegbreit, B, "Studies in Extensible Programming Languages" Ph.D. Thesis, Harvard University, May 1970.

[23] Wegbreit, B, "The ECL Programming System," Proc. AFIPS 1971 FJCC, Vol. 39, AFIPS Press, Montvale, N.J., pp. 253-262.

[24] Wegner, P. "Data Structure Models for Programming Languages," in Tou and Wegner, pp. 55-82.

[25] Weizenbaum, J. "The Funarg Problem Explained," M.I.T., Cambridge, Mass., March 1968.

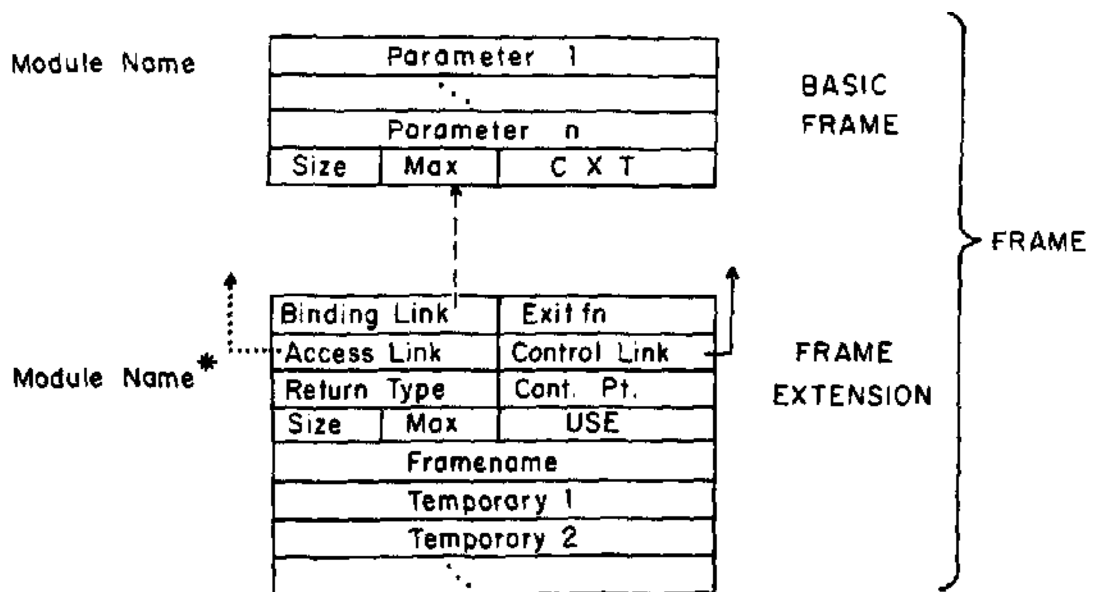

FIG. 1 GENERAL FRAME STRUCTURE

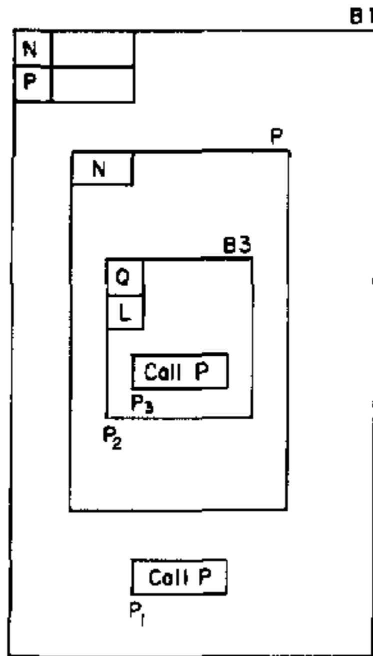

FIG. 2a (from Johnston)
BLOCK B1 WITH LOCALS N, P
PROCEDURE P WITH NEW VARIABLE N
BLOCK B3 WITH LOCALS Q, L
CALLS TO P WITHIN B1 AND B3

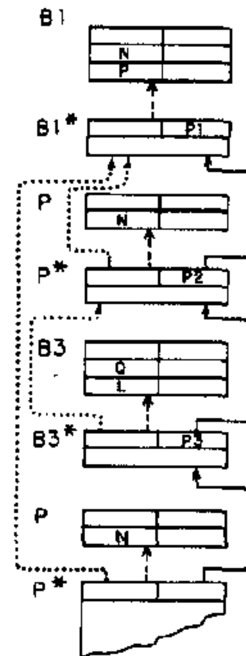

FIG. 2b SNAPSHOT OF FRAME STRUCTURE STARTING AT B1, CALL TO P, ENTER B3, CALL TO P

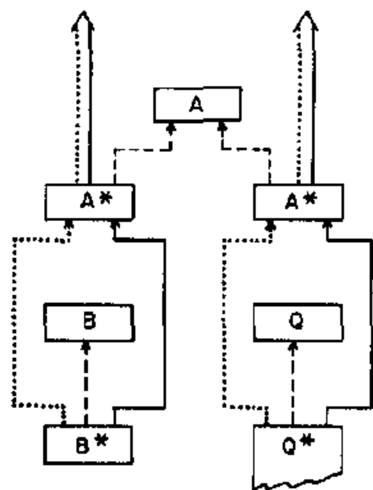

FIG. 3a COROUTINES SHARING ANCESTOR MODULE A, Q IS ACTIVE

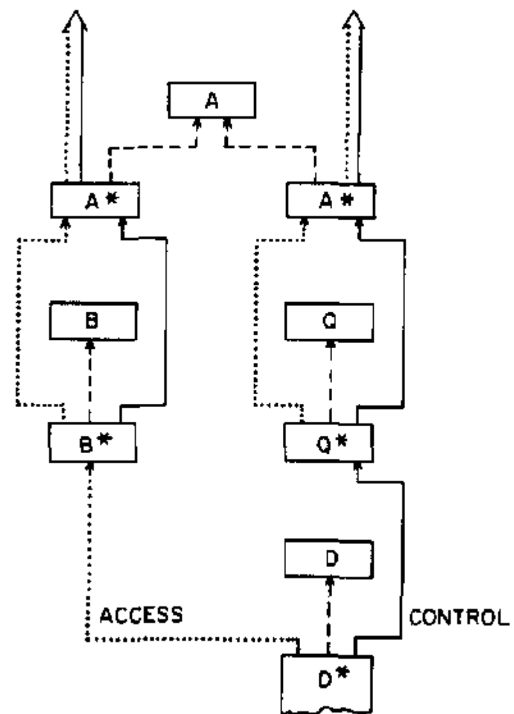

FIG. 3b COROUTINE Q EVALUATING FUNCTION D IN ACCESS CONTEXT OF B*

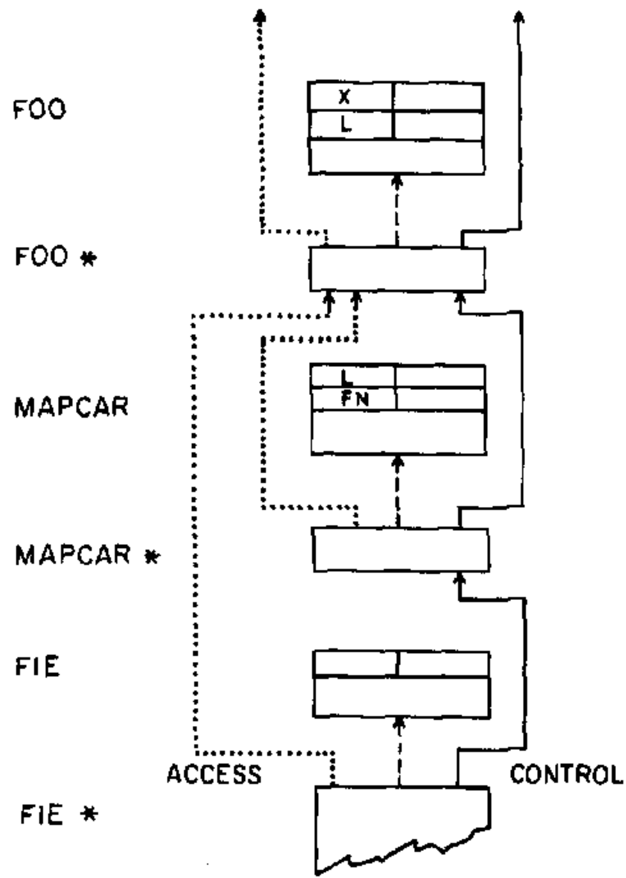

FIG. 4 APPLICATION OF A FUNCTIONAL ARGUMENT