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, FUNARG-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.I. 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-expressions, 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,

$$R(x,y) \;\Lambda\; R(y,z) \;\supset\; R(x,z)$$

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 y; 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 program 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. In programming systems terms, we want to compile the axioms, and do as many decisions as possible at compile-time.

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

$$R(a,b)$$

and if it asked the "question" $\sim R(b,c)$, it will generate the sub-question $\sim R(a,c)$. 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 R̄, use
lambda (y,z) begin local x;
             determine x from R(x,y)
             return sub-question ~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-determinism 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 conversion 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 "4" against "plus(x,1)" 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 $R_2(x)$. For example, the same transitivity axiom for R also calls for the following operator:

```
On a sub-question with R2, use
lambda (x) begin local y;
           determine y from R(x,y);
           return sub-question R2(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 R2, use
lambda (x) begin
           return sub-question P2(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 R2,   use
lambda (x) begin local y,z;
           determine y from P(x,y);
           return sub-question Q2(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 R2, 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 R2, 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 R2, 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 is? 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 picking 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 predicate-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
$s_1, s_2, \cdots s_{k-1}, s_k, s_{k+1}, \cdots s_n$
where s_k is the controlled statement.

(4) Construct a new operator where the statements are
$s_1, s_2, \cdots s_{k-1}, s_k*$
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) ∧ Q(x,y) ⊃ 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) ∧ Q(y,z) ⊃ 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 procedure
                      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 g, 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 Q_2 in one particular use of the above operator. Clearly g' 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) question, g is given as "return answer".

Discussion of applicability of the method. This procedure works in all cases where the non-deterministic 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 ly 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, René 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. Rulifson 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)

8. B. Raphel
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 al.
Common Base Language
Norwegian Computing Center, Oslo, 1970

11. R.M. Burstall et al.
Programming in POP-2
Edinburhg Univ. Press, 1971

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

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

14. 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