A PROGRAMMING TOOL FOR MANAGEMENT OF A PREDICATE-CALCULUS-ORIENTED DATA BASE*

Erik Sandewall
Uppsala University
Uppsala, Sweden

Abstract

This paper describes a LISP program, called PCDB, for storage and retrieval in a data base of predicate calculus (PC) formulas. The PCDB pacxage uses standardized representations of PC formulas, where ground unit clauses are stored efficiently, e.g. on the property-lists of their arguments, and other clauses have other representation. The major part of the PCDB package is a function generator, which accepts declarations of PC relations and functions, and Which also accepts "rules" (= non-ground axioms intended for use in deduction)

Declarations and rules are used to generate efficient code for storage and retrieval of "facts" (usually = ground unit clauses) in and from the data base. This generation process may be characterized as a "compilation" of the rules (from predicate calculus to LISP).

Key words and phrases

Axiom compilation, deduction, function generator, LISP, partial evaluation, retrieval.

Motivation

Many programming tasks in artificial intelligence require the manipulation of small data bases. Question-answering programs maintain a data base of accumulated facts, and inquire the data base for answers to questions. Robot programs maintain a data base which describes the robot's environment. Advanced CAI programs maintain a data base of the subject-matter that they have to teach. Simulated scientists (e.g. the Dendral program) maintain a data base of know-how in a narrow but deep field of human knowledge. These programs, and other similar ones, need to perform complex retrieval operations (and sometimes, reorganization) in the data base.

(x) This research was supported in part by the Swedish Natural Science Research Council (contract Dnr 2654-3)

In the design of such programs, it is crucial to find a good representation of one's data. Programming languages like LISP, PL/I, or SIMULA 67 offer a machine-oriented data structure; for example, LISP offers the use of property-lists, and SIMULA 67 offers the use of Hoare's record structure. However, it is often desirable to find a more problem-oriented representation, such as nested trituples, colored graphs, or "semantic nets".

Predicate calculus notation is often used as such a "high-level data language". The SRI A.I. project uses predicate calculus for their robot project (see Nilsson, 1969) and for question-answering (see Green, 1969). Two different methods for using predicate calculus as a data language in program manipulation programs have been proposed by Burstall (1969) and by Manna and Waldinger (1970). Burstall (1970) has proposed a set or conventions for expressing the deep structure of natural language in predicate calculus, and a similar proposal has been made by the present author (Sandewall, 1970). The list can be continued.

Predicate calculus offers the user two major advantages: First, it is a versatile notation, and second, the user can express the logical properties of his data through a set of axioms. These axioms can be used as a theoretical basis for the program, or they can be given to a theorem-proving program (such as QA3, see Green, 1969). In the latter case, the theorem-proving program "interprets" the axioms. This was discussed in Green, op.cit., page 131. In the former case, the same axioms have been manually re-written ("compiled") into a program. Using a theorem-proving program is more convenient, but a tailor-made program can be expected to be much more efficient.

Topic of this paper: the PCDB package

This paper describes an existing LISP "program" (or to be precise, function package) which combines the generality and convenience of a theorem-proving program with the relative efficiency of tailor-made programs. Our program is

called PCDB (for "Predicate Calculus Data Base"), and it is a function generator. PCDB takes the following types of input:

declarations of the predicate calculus relations and functions that the user wants to utilize;

facts, i.e. ground unit clauses which are intended as contributions to the data base;

questions, i.e. unit clauses which are to be proved from the data base. Question clauses may be ground (for closed questions) or non-ground (for open questions);

rules, i.e. non-ground (and usually non-unit) clauses which are to be accumulated, and later used for forward deduction from facts and/or for backward deduction in answering questions.

When the package receives rules, it generates "code" (i.e. LISP S-expressions) that corresponds to these rules (i.e. it "compiles" the rules), and when it receives facts and questions, this code is utilized. Declarations set certain flags and properties which govern subsequent code generation.

The PCDB package is useful in problem environments where there is a large and open-ended data base of "facts" (in the above sense of the word) and a relatively small and limited number of "rules". Question-answering applications and Burstall's notation for program descriptions satisfy this requirement. - The package is not useful in problem environments where the major part of the data base consists of "rules". Manna's approach to program analysis is an example of such an environment. For such applications, conventional resolution programs seem more suitable.

Example of predicate calculus usage

Before we proceed to a detailed description of PCDB, let us give an example of the use of predicate calculus as a high-level data language. With this example, the reasons for the design of PCDB will be more transparent.

Suppose we want to perform that standard exercise: writing a kinship handling, question-answering program. Some simple kinship relationships may be expressed in predicate calculus as

Sibling(Jesper,Bodil)
Male(Jesper)
Father(Jesper) = Edvin
Wife (Edvin) = Edla

etc. ("Sibling" stands for "brother or sister"). For simple facts there is no essential difference in effort between making these predicate-calculus statements and making the obvious property-list storage instructions. However, in more complex expressions, like

Sibling(Father(Hedvig), Wife(Neighbor (Halvard)))

(for "Hedvig's father is a sibling of Halvard's neighbor's wife) the predicate calculus formulation is probably more convenient than what we could immediately do with e.g. property-list storage instructions. In this sense, the notation is "high-level" and "problem-oriented".

Moreover, predicate calculus permits us to state general axioms, which characterize these kinship functions and relations, e.g.

$(x)\ \ Male(x) \equiv \neg Female(x)$

$(x)\ \ Child(x,Father(x))$

$(x)(y)(z)\ \ Child(x,y) \wedge Child(x,z) \supset y = z \vee Sibling(y,z)$

If we write a direct program for the kinship exercise (without the support of any standard program), then the information contained in these axioms must somehow go into that program. Normally, it goes into the retrieval part, but some of the axioms could also go into the storage part. For a trivial example, the first axiom could correspond to a segment of the retrieval procedure which says "if it has been asked whether x is male, and if there is no immediate information in the data base saying that he is or isn't, then ask as a sub-question whet er x is female, and negate the answer", and to a corresponding segment for the case where x is female. But alternatively, it could correspond to a section of the storage procedure which says "if you have to store that x is male, then store also that x is not female" and similarly for the symmetric case. The retrieval procedure does backward deduction, whereas the storage procedure does forward deduction. It is one purpose of PCDB to generate code for and to administrate backward and forward deduction. With this example as a background, let us now describe the PCDB program by tracing what happens when some selected expressions are given as "input", i.e. as argument to functions in the package.

Declarations of relations

LISP atoms areused for relations symbols, function symbols, and object symbols. The purpose of declaring a relation is to provide it with a number of functions (lambda-expressions) which are properties on the property-list of the relation. For example, a binary relation Father

(such that Father(x,y) means "y is the father of x") might be associated with at least the following lambda-expressions:

indicator	property
STOREDEF	(LAMBDA (X Y) (PUTPROP X Y 'FATHER) (ADDPROP Y X 'REVFATHER])
TESTDEF	(LAMBDA (X Y) (EQ X (GET X 'FATHER])
FETCHDEF	(LAMBDA (X) (GET X 'FATHER))

These properties are used for assertions, for closed questions, and for open questions, respectively. Additional properties are needed for deduction, for undoing previous assertions, etcetera. It would have been possible to design the PCDB package so that all of these properties are generated and assigned at the point where FATHER is declared to be a binary relation. However, we have preferred to do things as follows in PCDB:

at declaration time, some flags are put on the property-list of the new relation. These flags correspond directly to the information in the declaration. No functional expressions are generated.

when a functional property is needed, the system asks for it by doing e.g.

getd fill FATHER, TESTDEF]

where the function getdflt is defined essectially as

getdflt[a,i] = get[a,i] V putprop[a,i[a], il

The first time the TESTDEF property of FATHER is asked for, the value of get[FATHER, TESTDEF] is NIL, so testdef[FATHER] is evaluated, stored as a property, and returned as a value. This is the required function definition. In other words, the functions testdef, storedef, etc. all return lambda-expressions as values.

On each succeeding call for the TESTDEF property, it can be retrieved from the property-list, and it need not be computed again. (The value of the function testdef is memoized).

This design has several advantages:

(a) functional properties are not generated unless they are really needed;

(b) the user may make several declarative statements, provided they have all been given before the first time the relation (or function) is used;

(c) declarations may be implicit. Functions like testdef must clearly inquire the property-1ist of their argument for the flags that were assigned by the declaration. If these flags are not present, then testdef (and similar functions) can compute default values.

In fact, this is a simple example of backward deduction.

The PCDB package contains various service functions for doing declarations in a convenient way. However, let us ignore the conventions of those service functions, and look at their effect in the example of the relation Father, where the declaration might have immediate effect of putting the following properties of the property-list of the atom FATHER:

indicator	value	meaning
RELTYPE	CLEANPRED	binary relation; both arguments are "proper" (i.e. either argument may be omitted in open questions)
ARGTYPES	((AA) (AA))	both arguments must be represented by alphameric atoms
ONE-MANY	(MANY ONE)	relation is functional in second argument (i.e. nobody has more than one father) but not in first
LOCATED	ARGS	stored on the property-lists of the arguments

The functions testdef[r], storedef[r], etc. have been defined so as to branch on get[r,KELTyFE], get[r,LOCATED], etc. and to return a functional expression in each case. With the given declarations, the above-mentioned lambda-expressions will be obtained.

The PCDB package recognizes different kinds of relations, with different numbers of arguments (one, two, three, or more), different storage conventions (on property-lists of arguments, on property-list of relation symbol; hashed in an array is planned, etc.) and other idiosyncracies. Since each relation has its own storage

and retrieval routines, it becomes possible to assign tailor-made routines automatically to all relations.

The path of an assertion

Let us continue the example with the relation Father. The PCDB package contains a standard function assert, where evaluation of

```
assert[FATHER,DICK,JOHN]
```

will cause the following flow of control:

1. assert checks the well-formed-ness of the arguments, and calls

```
sysassert[ 1,  (FATHER  DICK  JOHN) ]
```

The number 1 is inserted as a default value, and limits the number of H-expressions that are generated in forward deduction (see below). It could have been given explicitly as a first argument to assert.

2. sysassert is defined in principle as follows:

```
sysassert[n,guc]  =  if test[guc] then T
                                  else
   prog2[ store[guc], apply[get[car[guc],
          ASSERTDEF],[cdr guc]]]
```

with

```
store[guc]  =  apply[ get[car[guc],STOREDEF],
                      cdr[guc]]
```

and

```
test[guc]  =  apply[ get[car[guc],TESTDEF],
                     cdr[guc]]
```

Thus we first use the TESTDEF property of the atom FATHER as a function, with the list (DICK JOHN) as an argument list. Normally, the relationship was not known to the system before. It is then stored by the STOREDEF property. Finally sysassert applies the ASSERTDEF property of the atom FATHER to the list (DICK JOHN). The ASSERTDEF property has in principle the form

```
(LAMBDA  (X Y)  (PROGN
   (PROG  ... )
   (PROG  ... )
    ...
   (PROG  ... )  ))
```

where each prog-expression is a compiled axiom in forward usage. (In some cases, the system may have been able to simplify the expressions so much that the prog vanishes). Initially, the ASSERTDEF property has the form

```
(LAMBDA  (X  Y) (PROGN))
```

and it is then updated each time an axiom is given to the system. Compiled forward axioms call sysassert with their conclusion, so triggering may be continued recursively. There are other functions which are similar to assert, but which do slightly different things. For example, the function claim makes a search to check that the new fact does not contradict the old data base, before it stores the new fact and triggers side-effects.

A user may often want to call an arbitrary LISP function during forward deduction, and to write e.g. the "rule":

$$P[x,y] \supset print[x]$$

This is easily done in PCDB by assigning a suitable STOREDEF property to the atom PRINT, or by writing the rule as

$$P[x,y] \supset Do[print[x]]$$

where the STOREDEF property of the atom DO is simply

```
(LAMBDA  (X)  X)
```

Relations used in backward deduction axioms can be defined similarly. This is equivalent to the "predicate evaluation" feature in various resolution programs.

The path of a closed question

Closed questions are represented as ground unit clauses ("guc"-s), just like facts. The following standard functions are presently provided in PCDB for answering closed questions:

test, which simply checks whether the clause has been explicitly stored

search, which does a breadth-first search through backward deduction axioms in order to prove the clause. There is a call to test at each tip of the search tree. Multiple occurrences of a sub-question are recognized.

recsearch, which is like search except that the search is depth-first, and multiple occurrences of sub-questions are not recognized.

ask, which calls search with its argument and with its negated argument, and then answers YES, NO, or DON'T-KNOW as described by Palme (1971).

The function test is similar to store, and both search and recsearch are similar to assert. There are functions syssearch, sysrecsearch, etc. and relations have function definitions on their property-lists under the indicators TESTDEF, SEARCHDEF, etc. For example, sysrecsearch is in principle defined as

```
(LAMBDA  (N GUC)
  (COND  ((TEST GUC)  T)
         ((ZEROP  N)  NIL)
         (T  (APPLY  (GET (CAR GUC)
                          'RECSEARCHDEF)
                     (CONS N (CDR GUC]
```

Moreover, the RECSEARCHDEF property of a binary relation has the form

```
(LAMBDA  (N*  X  Y)  (OR
  ..
  (PROG  ...
     [COND  ((SYSRECSEARCH (SUB1 N*)
             (sub-question))
              (RETURN T]
          ... )
  ...    ))
```

Each argument to OR is a compiled axiom, and each compiled axiom calls sysrecsearch once for each sub-question it generates.

SEARCHDEF properties (which are used by syssearch for breadth-first search) are similar to RECSEARQiDEF's, but instead of calling sysrecsearch, they store the sub-question on a FIFO subquestion queue which the function syssearch maintains as the value of a prog variable, which then is free in the SEARCHDEF.

Compilation of rules

The compilation process starts with a rule (i.e. a non-ground, and usually non-unit clause), and ends when this rule has been transformed to LISP code and inserted in an ASSERTDEF, SEARCHDEF, and/or RECSEARCIDEF property. Compilation is done in several passes. Let us illustrate it with a concrete example, namely the axiom

$$R[x,y] \wedge P[y] \wedge S[y,z] \supset Q[z,x]$$

Let us assume, furthermore, that this rule is to be used for forward deduction, so that when R[x,y] is asserted, and Ply] has previously been asserted, then for all z such that S[y,z] has been asserted, Q[z,x] is to be asserted. The treatment of rules for backward deduction is analogous.

Pass I: External to internal notation. This step transforms infixes (such as A, or infix relations) to prefix form. It also makes a list of the variables, and classifies the literals as an antecedent, a list of conditions, and a consequent. The result is

```
((X  Y  Z)                    variables
 (R  X  Y)                    antecedent
 ((P  Y) (S  Y  Z))           conditions
 (Q  Z  X))                   consequent
```

In this example, R(x,y) is selected as the antecedent, since it is to be the triggering literal (i.e. the compiled axiom is to be on the ASSERTDEF property of R). In general one may want to have several versions of the same axiom, with different choice of an antecedent.

Pass 2: Convert to PROG-expression. This pass generates a very inefficient but yet executable prog-expression, and can be considered as the compilation proper. In our example it will generate the expression

```
(R  (N X Y) (PROG  (Z ZQUE)
  B1  (BIND NIL '(P Y) 'B2 'END 2)
  M1  (CONT NIL '(P Y)  'END)
  B2  (BIND '(Z) '(S Y Z) 'B3 'M1 2)
  M2  (CONT '(Z) '(S Y Z) 'M1)
  B3  ((LAMBDA (X*) (SYSASSERT N (CLOSE X*)))
         '(Q Z X))
      (GO M2)
  END  (RETURN) ))
```

Thus the antecedent determines the contents of the first line (intended as lambda-variables etc.); each condition generates a bind-cont pair, and the consequent is used as an argument to the lambda-expression. The functions in this prog are defined as follows:

bind[varlist,literal, oontlabel, backuplabel, depth]. If varlist is NIL, then bind does a recsearch on the literal with the indicated depth. If the value is T (i.e. if the condition is verified), then bind does a goto the contlabel, where the next condition is tested, otherwise it does a goto the backuplabel, i.e. the corrt-statement of the previous condition.

If the varlist is non-NIL, then bind binds (using the LISP function set) the variables on the varlist. In general, the varlist contains all variables which occur in literal, and which have not occurred in the antecedent or in any previous literal. In position B2 above, bind will assign to ZQUE the list of all z which satisfy S(y,z) for the given y, and to Z the first element of ZQUE. Moreover, bind will go to contlabel if the value of zque is non-NIL (i.e. some z has been found)), and to backuplabel otherwise.

cont[varlist, literal, backuplabel]. This function has the primary purpose of doing (in the above example)

```
Z   := car[zque]
ZQUE := cdr[zque]
```

When zque has been exhausted, cont goes to backuplabel (where an "earlier" variable will be cont-ed), otherwise it leaves control to the next statement in the prog (where the next condition is bind-ed, or the consequent of the clause is processed). Moreover, if the varlist is empty, the corresponding bind only served as a test, and then cont should trivially go to backuplabel.

The reason cont is given literal as an argument is that in cases like

(CONT '(Y) '(R X Y) 'M6)

if R has been declared to be functional in the second argument, it is obvious that there is nothing to continue with. Then cont can again trivially go to backuplabel.

close[form]. This function is defined as

cons[car[form],evlis[cdr[form]]]

In the example, it is used to pick up the current values of *x* and *z*. - it should also be remarked that the variable n is used to control the number of new nodes that are introduced. N is decremented when we have forward axioms like

P(x,y) ⊃ R(x,f(y)

where y is a function from the predicate-calculus viewpoint.

Pass 3: Partial evaluation. This is an equivalence transformation which transforms the prog-expression into another expression, which has the same value and the same side-effects for all arguments, but which runs more efficiently. Partial evaluation is possible since all arguments to bind and cont have been given explicitly. For example, the function cont is defined as

```
cont[vl,e,m] =
  if null[vll then goto[m]
  elseif null[cdr[vl]] then
    (if . . . then goto[m]
     else prog2[ set[car[vl],car[eval[que[car
                                    [vl]]]]]]
                set[que[car[vl]],cdr[eval
                                [que[car[vl]]]]]]  )
  else ...
```

where que is a packing and memoizing function defined so that

que[Z] = ZQUE

In partial evaluation, the definitions of cont, que, etc. are inserted in the prog-expression; lambda-expressions are expanded, and function expressions are collapsed whenever possible, e.g. so that

(CAR (CONS x y)) -> x
(EVAL (QUOTE X)) -> x
(GOTO (QUOTE D) -> (GO 1)
(SET (QUOTE x) y) -> (SETQ x y)

With such simplifications, the expression in position B2 above is reduced to

```
B?  (COND ((NULL ZQUE) (GO MI))
          (T (SETQ Z (CAR ZQUE))
             (SETQ ZQUE (CDR ZQUE]
```

which is quite reasonable code.

Pass 4: Prog-reduction. Partial evaluation will leave a lot of redundant go statements in the *prog,* e.g. like

```
   (GO L)
    •*•
L (GO M)
    ••»
```

In this pass, the obvious simplifications are made.

In a fifth step, the prog-expression is inserted into the relevant ASSERTDEF or other ...DEF properties.

It should be noticed that pass 2 and step 5 are the only steps that are logically needed. Pass 1 only provides some added notational convenience, and is dispensable. Passes 3 and 4 serve to speed up the program, but they are not necessary For testing purposes, it is sometimes better not to use them.

Predicate-calculus functions

Operators which are functions from the predicate calculus viewpoint (these must be carefully distinguished from functions in the LISP system, which in a certain sense interprets the predicate calculus expressions), are handled in the following fashion:

Every PC function is assigned a LISP function definition, which follows a standard pattern. If *g* is an n-ary PC function, then it is also an n-ary LISP function which returns an expression of the form

(G8 G A B)

where G8 is a gensym-atom, and A and B are the evaluated arguments to g. This expression is called an H-expression. (H stands for Herbrand). It is generated the first time g[A,B] is evaluated, and the same (with eq) expression is obtained on each successive evaluation of g[A,B].

The atom G8 carries a property-list, on which relations can be stored. Moreover, the property-list of the first argument (in this case, A), contains a pointer to the above expression under a certain indicator. Notice that A need not be an atom, but it may itself have been generated by a PC function.

When functions appear in axioms, then linear code for the matching ("unification") is generated at compile time. For a trivial example, if the axiom

$$P[g[x,y]] \supset Q[y,x]$$

is declared for forward usage, then it is re-written in an early compilation step as

$$P[z] \wedge GG[z,x,y] \supset Q[y,x]$$

where gg is considered as a relation which has a peculiar mode of storage, but which is otherwise similar in all respects to user-declared relations. If the user declares g as a PC function, then the necessary declaration of gg is implicit. - Notice that this axiom would not be re-written if it were declared for backward usage. Thus new H-cxpressions may be introduced during deduction.

Comparisons with some existing programs

Several existing programs show similarities with PCDB in some respects. We shall attempt a comparison, even if there is on obvious risk that we are misinformed about some aspect of the other programs. The reader should take the discussion here as a first approximation.

QA3 and QA3.5 (see Green, 1969, and Garvey and Kling, 1969). QA3 is a resolution-oriented theorem-proving program, which maintains a "memory" (set of clauses that are stored in the system) and a "clauselist" (set of clauses that are active during a deduction). Clauses in memory are indexed by predicate letter and then by length; clauses on the clauselist by length only. Moreover, there are special heuristics for handling e.g. sequences of resolutions with binary clauses ("chaining"). QA3. is similar in all of these respects. The major differences between QA3 and PCDB are:

1. QA3 interprets all "rules"; PCDB compiles them.

2. QA3 can perform resolution proofs according to arbitrary strategies. The proofs performed by the PCDB search executive and the compiled axioms correspond to some highly restricted resolution strategies.

3. QA3 treats all clauses uniformly, and does not store ground unit clauses ("facts") in a special way as PCDB does, (it is possible to interface QA3 with a program for such storage, for example by using the predicate evaluation feature. However, it seems to be a non-trivial task to make this work fully automatically, and to implement it for clauses where the relation's arguments have been constructed with PC functions).

The difference indicated in point 2 implies that QA3 is more general-purpose than PCDB is, and that it is more suitable for tasks where a major part of the clauses as non-ground and non-unit. On the other hand, points 1 and 3 enable us to guess that PCDB should be much faster in those cases where it is applicable, although no comparative measurements are yet available.

The NIH Heuristics Laboratory program (see Norton, 1971, and Dixon, 1971). This program is similar in approach to QA3. In some experiments, the NIH program has been guided by heuristic search, but this does not affect the comparison with PCDB. The NIH group have compiled axioms using partial evaluation, and did so a year before us. However, their compilation (at least as described in the paper by Dixon) seems to be more restricted. It is performed so that

$$\text{resolve}[c,c'] = (\text{compile}[c])\ lc']$$

whereas the execution of one compiled rule in PCDB may be equivalent to a large number of resolutions.

Planner (see Hewitt, 1970). Like PCDB and unlike the preceeding programs, Planner is a substrate for special-purpose theoreni-provers, rather than a general-purpose theorem-prover. Planner is also more data-base-oriented than the preceeding programs. For example, Planner uses the idea of associating procedures with relation symbols, and to execute these for storage and/or retrieval . The dichotomy between forward and backward deduction is present in Planner, where one speaks about "consequent" and "antecedent" theorems, re spectively.

The major difference is that Planner uses a general high-level control mechanism, which is similar in approach to non-deterministic programming. This control mechanism is available to the user of Planner; the user of PCDB does not have it (except trivially when he can utilize the system's search executive for backward proof in some round-about way). The control mechanism is also used inside Planner. The occurrences of this correspond e.g. to the breadth-first search that is performed by the function syssearch in PCDB, and to the loops

that are generated by bind and cant in compiled axioms. This difference is correlated with another:

Planner thinks about its user as a programmer, and about its "theorems" as programs, albeit in a very machine-remote programming language which assumes the above-mentioned control structure. PCDB thinks about its input as static information ("declarations", "facts", "rules"), plus some tags which hint when and where this static information is to be used; and PCDB takes as its task to "digest" its input so that it can later be used in the indicated situations. (This attitude of PCDB can be overcome, e.g. using the function do that was described above, but it is still significant). As a consequence of this, PCDB assumes predicate calculus notation as input, whereas Planner assumes its own input language. (It is an interesting problem to write a translator from predicate-calculus-plus-control-tags to the Planner language). Thus PCDB is in several respects slightly closer to conventional theorem-proving programs than Planner is.

Present status and planned extensions of PCDB.
Most features described in this paper are presently working (May 20, 1971). The exceptions are: (a) pass I of compilation (infix to prefix notation); (b) pass TV of compilation (prog-reduction) and the function collapsing step in pass III (partial evaluation); (c) function-to-relation conversion in compiling rules with PC functions in them. The immediate plans are to complete these steps; to add a package for answering open questions (exists in outline); and to compare the efficiency of PCDB-generated code with manually produced code.

Acknowledgements

Many thanks to Lennart Drugge, Anders Haraldson, Rene Reboh and Arne Tengvald, who helped greatly with testing and debugging various parts of the PCDB program. Thanks also to David Luckham, who patiently explained some of the intricacies of resolutionology, and to members of the SRI A.I. group for the opportunity to discuss and to play with the QA3.5 program.

References

Burstall, R.M. (1969)
Formal description of program structure and semantics in first-order logic
in Meltzer & Michie (eds) Machine Intelligence, Vol. 5 (Edinburgh, 1969)

Burstall, R.M. (1970)
Formalising the semantics of first-order logic in first-order logic, and an application to planning for robots

Dixon, John K. (1971)
THE SPECIALIZER: A Method of Automatically Writing Computer Programs
NIH Heuristics Laboratory (unpublished)

Garvey, Thomas D. and Kling, Robert E. (1969)
User's Guide to QA3,5 Question Answering System
SRI Artificial Intelligence Group, Technical Note 15 (1969)

Green, Cordell C. (1969)
The application of theorem proving to quest ion-answering systems
SRI Artificial Intelligence Group, June 1969

Hewitt-, Carl (1970)
PLANNER: a Language for Manipulating Models and Proving Theorems in a Robot
MIT Project MAC Artificial Intelligence Memo No. 168

Manna, Zohar and Waldinger, Richard J. (1970)
Towards Automatic Program Synthesis
Stanford Artificial Intelligence Project Memo AIM-127

Nilsson, Nils (1969)
A mobile automaton: an application of artificial intelligence techniques
Paper presented at the first International Joint Conference on Artificial Intelligence (1069)

Norton, Lewis M. (197])
Experiments with a Heuristic Theorem-Proving Program for Predicate Calculus with Equality
NIH Heuristics Laboratory (unpublished)

Palme, Jakob (1971)
Making Computers Understand Natural Language
in Findler (ed.), Artificial Intelligence and Heuristic Programming
Oxford University Press, 1971

Sandewall, Erik (1970)
Representing natural-language information in predicate calculus
in Meltzer & Michie (eds) Machine Intelligence, Vol. 6 (Edinburgh, 1970)