Translating
Annotated Predicate Calculus
to PROLOG

Kah-Eng Pua
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ISG 84-1

June, 1984
I would like to thank Professor Ryszard Michalski for his support and guidance. From him I learned much about the ways of research and am most grateful. I would also like to thank Jeff, Kaihu and Tony for their many useful suggestions. Finally, many thanks should go to Lai-Choi for her encouragement.
1. Introduction

The main purpose of this paper is to provide an illustration of the process of translating Annotated Predicate Calculus (APC) to PROLOG. APC is an invention of Professor Michalski. It is built on the basis of First Order Predicate Calculus (FOPC) with some additional features that make it especially suitable for knowledge representations in machine learnings, expert systems and some other fields in the realm of Artificial Intelligence. The main difference between APC and FOPC can be summarized as follows:

1. Each descriptor (term or predicate) in APC is assigned an annotation. The annotation stores background knowledge about the descriptor. It specifies the type and the domain of the descriptor, the relationship between the descriptor and other descriptors, the type and the number of its arguments, constraints that the descriptor must satisfy, and any other relevant information.

2. Logical operators " and v apply not only to predicates, but also to terms. This feature makes APC expressions even closer to natural language.

3. Predicates that express relations =, /=, >, >=, <, <= are expressed explicitly, using infix notations, as relational statements called selectors.

4. Existential and universal quantifiers are generalized into numerical quantifiers. With the introduction of numerical quantifiers, definite numerical propositions such as "There exists one book on my desk." can be expressed with great ease.

Details of APC and its applications are found in [Michalski 1983].

A program dealing with the translation of APC to PROLOG has been written in UNSW-PROLOG under UNIX system. This paper will provide a brief explanation to the ten steps used in the translation.
2. Notations

Let me first introduce the notations used in the program.

a. Logical operators

- negation

& conjunction

v disjunction

==⇒ implication

<=⇒ equivalence

\ exception

b. Binary relation symbols in selectors

.==. equal

./=. not equal

.>=. greater or equal

.<=. smaller or equal

>. greater

.< smaller

c. Relational operators in numerical quantifiers

< less than

<= less than or equal

> greater than

>= greater than or equal

d. Binary operator in numerical quantifier

.. range

e. Universal quantifier

all(X, p(X)) where p(X) is a formula
3. Steps

As mentioned above, there are ten steps in the program which translates APC to PROLOG. Before walking through each step, let me say a few words about the program in general. The formulation of the steps is based heavily on the description by W. F. Clocksin and C. S. Mellish in their book Programming in Prolog [Clocksin and Mellish]. Since APC is an extension of FOPC, some additional steps are added to the six steps described in the book. Also, in this program, cares are taken to ensure that all the ten steps are ordered in such a way that the efficiency of the program is maximized in the sense that transformation in each step is done once and for all without having to repeat certain similar transformations in the process of translation.

Let's walk through the program step by step.

3.1. Step 1: Changing Variables to #i's

In this step, the built-in procedure numbervars(X,N,Z) is called to change all the variables in the input APC expression to #i's, where i's are integers starting from N. After this step, each distinct variable in the input will be changed to distinct #i. The justification for this step is that it will prevent variables in input APC from matching undesirable expressions in later steps. For example:
exists([X1, X2], contains(X1, X2))

will be transformed to:
exists([#1, #2], contains(#1, #2))

3.2. Step 2 : Remove Implications and Transform Exceptions and Equivalences

A predicate rmimpl(X, Y) is defined here. X is the input and Y is the output of this step. There are three goals in this step:

First, expressions involving implication in the form \( a \implies b \) will be transformed to \( \neg a \lor b \).

Second, expression with exception symbol in the form \( a \not\rightarrow b \) will be changed to the form \( (a \leftrightarrow \neg b) \land (\neg a \leftrightarrow b) \).

Third, expression with equivalence symbol in the form \( \neg(a) \leftrightarrow b \) or in the form \( a \leftrightarrow \neg(b) \) will also be converted to \( (a \leftrightarrow \neg(b)) \land (\neg(a) \leftrightarrow b) \) since there are logically identical to exception relations. Also, expressions with equivalence symbol in the form \( a \leftrightarrow b \) or \( \neg(a) \leftrightarrow \neg(b) \) will remain the same. The reason for keeping equivalence symbol is that in doing so, we can avoid producing Prolog statements which will easily go into an infinite loop when they are executed. For example, if an equivalence relation in the form \( a \leftrightarrow b \) is transformed into \( (a \implies b) \land (b \implies a) \) in this step, it will, after going through all the steps described in this paper, be translated to the following Prolog clauses:

\[
a : - b.
b : - a.
\]

which will easily cause an infinite loop in a Prolog program.

Keeping equivalence and exception relations in the forms depicted in the second and third goals above may require a separate Prolog program to interpret them, it is, however, much better than translating them into the "trouble causing" form and put the burden of avoiding the
infinite loops on the users. Two examples are given below to illustrate how this step works:

1. all(#1, man(#1) ==> mortal(#1))
2. state(on) - state(off)

They will be changed to:

1. all(#1, ¬(man(#1)) v mortal(#1))
2. (state(on) <==> ¬(state(off))) & (¬(state(on)) <==> state(off))

3.3. Step 3: Converting Numerical and Distinctive Quantifiers to Existential Quantifiers

Two predicates conv(X, Y) and convtdla(X, Y) are defined in this step. conv(X, Y) will convert each numerical quantifier to corresponding existential quantifier. convtdla(X, Y) will traverse the first argument, which is a list, of each distinctive quantifier and return the corresponding existential quantifier in "embedded form".

Let's take a closer look at the conversion of numerical quantifiers to existential quantifiers. There are altogether four types of numerical quantifiers. Each of them is illustrated as follows:

Type 1. An example of this type is:

exists( >6, #1, horse(#1))

It reads "There exists more than six horses." Notice that in this type, the relational operator preceding the integer can be chosen from any operator in 2c. After this step, we have the above expression converted to:

exists( #1, horse(#1) & number(#1, >6))

where number(X, Y) is a special predicate introduced in this step to carry the message given in the first argument of the numerical quantifier.
The introduction of \textit{number}(X, Y) does not merely provide a syntactical change to the original numerical quantifier. Although this paper is not concerned, in general, about machine inference on APC expressions, a brief discussion of the role of the predicate \textit{number}(X, Y) in the process of inference may be helpful in explaining why it is introduced here.

In a simplified version of an inference program that is being developed, two modes, assertion mode and query mode, are used to carry out the inference process. In assertion mode, every input APC expression will go through exactly the same steps discussed in this paper and its result will be loaded to Prolog. In query mode, however, each input question will undergo different transformations in the last two steps.

As an example, if we assert that "There exists six horses." and type to Prolog in assertion mode:
\begin{verbatim}
exists( 6 , X . horse(X) )
\end{verbatim}
after going through all the steps, we will have the following loaded to Prolog:
\begin{verbatim}
horse(skn1).
number( skn1 , 6 ).
\end{verbatim}
Now, if we want to ask "Is there more than five horses?" and type to Prolog in query mode:
\begin{verbatim}
exists( >5 , X , horse(X) )
\end{verbatim}
we will have the following query entered to Prolog:
\begin{verbatim}
horse(X) , num( X , >5 )?
\end{verbatim}
where \textit{num}(X, Y) is a predicate, defined in the inference program, which will look for the desired predicate \textit{number}(sk1, 6) and return the desired value. In this particular example, the answer to the above query is:
\begin{verbatim}
X = skn1
\end{verbatim}
**Type 2.** This type of numerical quantifier is similar to the one mentioned above. The only difference is that the first argument is an integer without any relational operator preceding it. For example:

```prolog
eexists( 6, #1, horse(#1) )
```
reads "There exists six horses." It will be transformed to:

```prolog
eexists(#1, horse(#1) & number(#1, 6) )
```

**Type 3.** The form of this type is shown by the following example:

```prolog
eexists( 2 .. 6, #1, horse(#1) )
```
It reads "There exists two to six horses." It will be transformed to:

```prolog
eexists( #1, horse(#1) & number(#1, 2 .. 6) )
```

**Type 4.** The first argument of numerical quantifiers of this type is an expression involving disjunction(s) and integers. For example:

```prolog
eexists( 2 v 6, #1, horse(#1) )
```
reads "There exists two or six horses." It will be transformed to:

```prolog
eexists( #1, horse(#1) & number(#1, 2 v 6) )
```

The conversion of distinctive quantifiers to existential quantifiers is exemplified in the following example:

```prolog
eexists( [#1, #2], contains(#1, #2) )
```
It is converted to:

```prolog
eexists( #1, exists( #2, contains(#1,#2) ) )
```

Note that #1 and #2 above will be converted to two distinct Skolem constants in a later step.
3.4. Step 4 : Rewriting Compound Terms or Compound Selectors

Two predicates, rewrite(X, Y) and re(X, Y), are defined in this step. rewrite(X, Y) goes recursively into the components of input formula until it reaches a term or a selector. It then passes the term or selector to re(X, Y). The task of re(X, Y) is to test if every term or selector passed to it is compound. If so, it will go recursively splitting the term or selector into simple terms or simple selectors respectively. For example:

\[\text{color}(\#1 & \#2) = (\text{red v blue})\]

will be transformed to:

\[(\text{color}(\#1).=\text{red} & \text{color}(\#2).=\text{red}) v (\text{color}(\#1).=\text{blue} & \text{color}(\#2).=\text{blue})\]

3.5. Step 5 : Bringing Negation Inwards

We defined two predicates in this step. The first predicate negln(X, Y) takes the input X and apply De Morgan's Law or rules of quantification negation to it until negations precede only terms or atoms. The second predicate neg(X, Y) will take the X and return \(^{-}\)X. Two examples are given below:

1. \(^{-}\)(\(^{-}\)(\text{horse}(\#1)) v \text{animal}(\#1))
2. \(^{-}\)(\text{exists}(\#2, \text{dog}(\#2) & \text{horse}(\#2)))

They will be transformed to:

1. \text{horse}(\#1) & \(^{-}\)(\text{animal}(\#1))
2. \text{all}(\#2, \(^{-}\)(\text{dog}(\#2)) v \(^{-}\)(\text{horse}(\#2)))

3.6. Step 6 : Dropping Existential Quantifiers

In this step, Skolem constants are introduced to instantiate the existential quantifications. There are altogether five predicates defined in this step. The first predicate dropexist(X, Y, Z) has three arguments. X is the input formula, whereas Y holds the result
obtained by Skolemising $X$. $Z$ is a list holding the scope of the universal quantifiers in which an existential quantifier occurs. $\text{genfun}(X, Y)$ causes $Y$ to be instantiated to a new atom built up from one of the constants, $skm$ or $skn$, and a number that has not been used before. If the existential quantifier is derived from a numerical quantifier in Step 3, then the constant $skn$ will be used; otherwise, $skm$ will be adopted. $\text{subst}$ and $\text{subst\_args}$ will replace all variables in existential quantifiers by Skolem constants obtained from $\text{genfun}$. $\text{append}(L_1, L_2, L_3)$ will append two lists $L_1$ and $L_2$ to form a new list $L_3$. Three examples are given below to show how this step works:

1. $\exists #1. \text{horse}(#1) \& (\text{color}(#1).=\text{red}) \& \text{number}(#1, >6)$
2. $\forall #1. \exists #2. \text{p}(#1, #2)$
3. $\forall #1. \forall #2. \exists #3. \text{p}(#1, #2, #3)$

The corresponding outputs will be:

1. $\text{horse}(skn1) \& (\text{color}(skn1).=\text{red}) \& \text{number}(skn1, >6)$
2. $\forall #1. \text{p}(#1, skm2(#1))$
3. $\forall #1. \forall #2. \text{p}(#1, #2, skm3(#1, #2))$

### 3.7. Step 7: Dropping Universal Quantifiers

By this step, the only quantifier left at the outside of the formula is the universal quantifier. We can just drop it without changing the meaning of the formula. The predicate $\text{dropall}(X, Y)$ defined in this step will do this job. For example:

$$\forall #1. \text{p}(#1, skm2(#1))$$

will be transformed to:

$$\text{p}(#1, skm2(#1))$$
3.8. Step 8: Distributing Conjunctions over Disjunctions

By this step, the only connectives left in the formula are v, & and ¬. We can now put the formula in conjunctive normal form as follows:

\[(p \& q) v r \text{ is changed to } (p v r) \& (q v r)\]
\[p v (q \& r) \text{ is changed to } (p v q) \& (p v r)\]

The two predicates, *distrib*(X, Y) and *distributing*(X, Y), will do this job. *distrib* will look for the disjunction and passes it to *distributing* which, in turn, looks for the two forms depicted above and transforms it to conjunctive normal form.

3.9. Step 9: Packing Conjunctions into Clauses

We define four predicates here -- *clausify*, *inclause*, *notin*, and *putin*. *clausify* will build up an internal representation of a collection of clauses. Each collection of clauses is a list which contains clauses represented in the form cl(A, B), where A is a list containing literals that are not negated and B is a list containing literals that are negated but without their negation signs. The result of *clausify* is returned through the second argument Y. *notin* prevent the same atomic formula from going into both A and B list. The reason is that in doing so we can avoid printing out, in a later step, trivially true statements such as:

\[\text{man}(\#1) :- \]
\[\text{man}(\#1).\]

which not only says nothing but may become a goal that can never be satisfied. The last predicate *putin* will place each new atomic formula into the corresponding list. Four examples are given below to show how this step works:

1. \((\neg(\text{cat}(\#1)) v \text{animal}(\#1)) \& (\neg(\text{animal}(\#1)) v \text{can}_\text{move}(\#1))\)
2. horse(skm1) v dog(skm1)
3. \((\neg(\text{horse}(\#1)) v \neg(\text{dog}(\#1)))\)
4. deaf(#1) v dead_man(#1) v ~( be_shouted_at(#1) ) v ~( no_response(#1) )

The corresponding outputs will be:

1. [ cl([ animal(#1) ], [ cat(#1) ]), cl([ can_move(#1) ], [ animal(#1) ])]
2. [ cl([ horse(skm1), dog(skm1)], [ ])]
3. [ cl([ horse(#1), dog(#1)] ]
4. [ cl([ deaf(#1), dead_man(#1)], [ be_shouted_at(#1), no_response(#1)] )]

3.10. Step 10: Printing Clauses

In this last step, six clauses are defined to print out the internal representation for the clauses built up in the previous step. printcl looks for the clauses in the input list, whereas printdis, printcon1 and printcon2 will print out disjunctions and conjunctions (i.e., the right-hand side of the clauses in Prolog) respectively. printchar and putout will print out each atomic formula character by character and at the same time check if the character #i is encountered. If so, printchar will convert it to Xi and print out the Xi. The outputs in the previous step are printed as follows:

1. animal(X1) :-
   cat(X1).
   can_move(X1) :-
   animal(X1).

2. horse(skm1) v dog(skm1).

3. contradiction(( horse(X1), dog(X1) )) :-
   horse(X1),
\[ \text{dog}(X1). \]

4.

\[ \text{deaf}(X1) \lor \text{dead_man}(X1) :\]

\[ \text{be_shouted_at}(X1), \]

\[ \text{no_response}(X1). \]

In example 2 and 4 above, \( \lor \) is used to take the place of "\( ; \)". This is because "\( ; \)" is predefined as a Prolog function and cannot stand alone as a predicate or appear at the left hand side of a clause. Again, in the inference program that is being developed, a Prolog predicate is defined to take care of the above two types of terms or clauses whose forms are generalized as follows:

\[ A_1 \lor A_2 \lor \ldots \lor A_n. \]

or

\[ A_1 \lor A_2 \lor \ldots \lor A_n :\]

\[ B_1, B_2, \ldots, B_m. \]

In example 3 above, the clause head \textit{contradiction} is used to take the role of the built-in function \textit{not}, which cannot appear at the left-hand side of a clause or be treated as a predicate. The design of the clause with clause head \textit{contradiction} enables us to check the possible inconsistency in the assertions in our input stream. For example, if we decide that "there should not exist in this world anything which is both a horse and a dog" and type in:

\[ \neg (\exists X, \text{horse}(X) \land \text{dog}(X)) \]

we will get the result shown in example 3 above. However, if later we somehow assert also that:

\[ \exists X, \text{horse}(X) \land \text{dog}(X) \]

we will get this time:

\[ \text{horse}(\text{skm2}). \]
dog(skm2).

The inconsistency in our input assertions can be found out by typing to PROLOG:

contradiction(X)?

and get the result:

X = horse(skm2), dog(skm2)

4. Future Work

While this report is being prepared for publication, another program dealing with
machine inference using APC expressions as input is in progress. So far, it has been able to carry
out inference on assertions expressed in standard FOPC and definite numerical propositions
expressed as numerical quantifications in APC. At present, efforts are being put in finding a way
to transform equivalent relations to Prolog (or, alternatively, a form that can be handled by the
inference program written in Prolog). This is a very difficult task that may need quite some time
to fulfill.

5. Conclusion

An illustration of a program which deals with the translation from APC to PROLOG
has been given in this paper. Owing to the fact that APC has certain features which add to the
expressive power of FOPC, the task of knowledge representation can be done in such a way that
it is much closer to human thinking than it would otherwise be done in the conventional FOPC.
This program may path the way for the realization of the more effective knowledge representation
of APC in the machine.
References


This paper will provide a brief explanation to the ten steps used in a program which translates Annotated Predicate Calculus to Prolog. Annotated Predicate Calculus is built on the basis of First Order Predicate Calculus with some additional features that make it especially suitable for knowledge representations in machine learning, expert system and some other fields of Artificial Intelligence.