PREDICATE LOGIC AS A COMPUTATIONAL FORMALISM

K L Clark

Research Monograph: 79/59 TOC

December 1979

Abstract

This monograph attempts to give a concise introduction to the theory and practice of logic programming. It has five chapters. The first is an introduction to the use of the Horn clause subset of predicate logic as a programming language. A non-deterministic abstract evaluator is defined and its implementation on a stack machine is briefly described. Chapter 2 is devoted to control issues. It introduces program annotations as a language for controlling the non-deterministic evaluator. Using annotations one can set up coroutining evaluations and make the order of evaluation of a Horn clause procedure dependent upon its mode of use. Chapter 3 reviews the model theory and fixed point semantics of Horn clause programs. It presents a new proof of the equivalence between the procedural and mathematical semantics of logic programs. Moreover the equivalence proved is stronger than the classical completeness result. The equivalence is stated in terms of the answer substitutions that are the output of an evaluation. In chapter 4 the mathematical semantics is used to justify various techniques for verifying logic programs. All of these consist of constructing around the logic program a first order theory about the relations it computes. Certain theorems of this theory are then correctness and termination results for the program. Finally, chapter 5 inverts the verification process and explores the use of a particular verification method, consequence verification, as a technique for deriving programs from their specifications. Starting from an axiomatisation of a relation in unrestricted first order logic it examines ways in which we can systematically search for a set of computationally useful Horn clause theorems. These are the logic program for the relation.

Keywords: logic programming, mathematic semantics, first order theories, verification, synthesis, program transformation, implementation of non-deterministic languages, logic + control algorithms.

Department of Computing
Imperial College of Science and Technology
University of London
180 Queen's Gate
London SW7 2BZ

Telephone: 01-589-5111 Telex: 261503

116pp

Contents

of LUSH resolution.

Chapter 4 is nearly all new material on the verification of logic programs, although it is a development of some earlier research. First, it is shown that we can express properties such as correctness and termination as first order sentences about the program computed relations. If the sentence is true, the program has the verification property it expresses. It is then shown how these sentences can be proved by deriving them as theorems of the theory of the program computed relations. The axioms of this theory are the clauses of the program strengthened to equivalences and induction schemas. The use of these stronger axioms is justified by appeal to the fixpoint semantics. Program verification techniques such as structural induction and computational induction become first order proofs using particular induction schemas. Finally, a quite different verification method is investigated. In this method the program clauses are not used as axioms; they are derived as theorems. This verification method can be interpreted as a generalisation of fixpoint induction.

If we can verify a logic program by confirming that each of its clauses are theorems, we can synthesise a logic program by finding its clauses as theorems. That is, we can start with an intuitively correct first order axiomatisation of some relation we want to compute. We can then try to piece together a logic program by deriving a set of Horn clause theorems. This deductive approach to the construction of logic programs is the subject of Chapter 5. A method for systematically searching for Horn clause theorems is described and exemplified. We then show how we can sometimes piece together quite different programs by taking different subsets of some set of theorems. Finally, we apply the program derivation techniques to the task of transforming a given logic program P. We treat certain axioms of the theory of the relations it computes as specification axioms implicitly given by the program. From these a different set of Horn clause theorems is derived, which is a new program P'.

Acknowledgements

Special thanks are due to Bob Kowalski who introduced me to logic programming. Without his encouragement and support I should never have started on this research.

I should also like to thank all the people with whom I have collaborated over the last four years on this and related research. Their influence has been considerable.

The text was produced using the editing and formatting software of the Computer Systems Research Group at Queen Mary College. Being able to use this was a great convenience, and it is much appreciated.

Chapter 1 Syntax and Procedural Semantics

In this chapter we survey the machine oriented aspects of using the Horn clause subset of predicate logic as a programming language. We describe a resolution theorem prover which, given a logic program and a computation rule, executes the program along the lines of a conventional program interpreter. It gives us a procedural semantics for logic programs. We briefly describe a stack implementation.

1.1 Inference as computation

Robinson's machine oriented resolution inference rule for the clausal notation of predicate logic [Robinson 1965] was the first essential step along the road to viewing inference as computation. Then Green[1969] showed that a resolution theorem prover could be used to 'simulate' a computation. However it was Hayes [1973] and Kowalski [1974,1979b] who gave resolution inference an explicit procedural interpretation, Kowalski for the Horn clause subset of predicate logic and Hayes, for the most part, for logic programs comprising a set of equality statements. The final credibility was provided by the Marseilles [Roussel 1975] and Edinburgh [Warren et alia 1977] PROLOG implementations. These are essentially computationally efficient Horn clause theorem provers. The Edinburgh implementation, which borrowed and extended the implementation techniques of the Marseilles PROLOG, is particularly impressive. It pre-compiles much of the work of a unification (the basic unit of computation of a logic program) as a sequence of machine level instructions associated with the program clauses. For list manipulation programs it compares favourably with compiled pure LISP; [Warren et alia 1977] has the details. PROLOG has also been implemented in Budapest[Szeredi 1977], Leuven[Bruynooghe 1976], London[Clark & McCabe 1979] and Waterloo[Roberts 1977].

We shall follow Kowalski and use Horn clauses as the basic programming notation. We assume some familiarity with the concepts of unification and resolution, however we shall give a brief explanation of these ideas.

1.2 Syntax of logic programs

DEFINITIONS
(1) A **Horn clause implication** is a sentence of the form

R(t1,..,tn)<-A1&...&Am.

Each Ai, like R(t1,..,tn), is an atomic formula. When m=0, and the antecedent of the implication is empty, we call it an **assertion**.

(2) Something is an **atomic formula** (or **atom**) if it is of the form R(t1,..,tn), n≥0, where R is an n-ary relation name (predicate) and t1,..,tn are terms.

(3) A **term** is a variable, a constant, or of the form f(t1,..,tn), n>0, where f is an n-ary function name (functor) and t1,..,tn are terms.

We assume a denumerable set of names comprising finite length strings over a finite alphabet which does not include the logical symbols "<-" "," "(" ")" and "&". Whether or not a name is a variable/constant, a functor or a predicate can be determined by context. So a functor is just a name used in the functor context, etc. To distinguish variables from constants we adopt the convention that names beginning with an lower case letter are varables. All other names in the variable/constant context are constants. Informally, we shall use infix notation for both terms and atoms. For example, we shall write "u in u.x" instead of "in(u,.(u,x))", and "tom married mary" instead of "married(tom,mary)". As with these examples, we shall underline the infix predicates. We shall assume that infix functors associate to the right, so "2.3.Nil" is syntactic sugar for ".(2,.(3,Nil))".

Declarative reading

If x1,..,xk are all the variables of the clause

R(t1,..,tn)<-A1&..&Am

we can read it as:

for all x1,..,xk, R(t1,..,tn) if A1 and A2... and Am.

Alternatively, if y1,...,yi are variables that only appear in the antecedent A1&...&Am, and z1,...,zj are the variables that appear in the consequent, we can read it as:

for all z1,...,zj, R(t1,..,tn) if
there exists y1,...,yi such that A1 and...and Am.

The alternative reading derives from the fact that a universally quantified implication

(∀x)[P<-Q]

is logically equivalent to

[P<-(∃x)Q]

when x does not appear in P. We use "(∀x)" when we want to explicitly indicate a universal quantification of a variable x, "(∃x)" for an existential quantification.

We shall say that the above clause is about the predicate R, because R

is the predicate of the consequent atom of the implication.

DEFINITION
A **logic program** is a set of Horn clause implications which contains at least one implication about each predicate appearing in any of its clauses.

As in all programming notations we shall assume that some predicates are primitive, with a fixed 'system' provided set of clauses. An example would be the product predicate. Atoms that use the predicate will usually be written as t1=t2xt3, t1,t2 and t3 being the terms of the atom and "..=..x.." being the infix relation name. Conceptually, we treat the system provided implementation as equivalent to the infinite set of assertions:

```
0=0x0<-
0=1x0<-
0=0x1<-
1=1x1<-
  .
  .
  .
4=2x2<-
  .
  .
```

Of course, in practice only a finite subset of these assertions will be accessible, that subset being represented procedurally so as to make use of the hardware operations of the machine.

example program - 1

```
append(Nil,x,x)<-                                    (2)
append(u.x,y,u.z)<-append(x,y,z)
```

is a logic program for the predicate "append". As the mnemonic content of this relation name indicates, the two statements of the program can be read as statements about appending list structures. We take "Nil" to be the name of the empty list, and "." to be the name of a list constructor such that u.x is the list x with u 'consed' onto the front. With this interpretation, the assertion of the program tells us that for all x, appending the empty list onto x leaves it unchanged. The implication tells us that for all x,y,z and u, if z is the result of appending x and y then u.z is the result of appending u.x and y.

Recursive data structures

A term of the form t1.(t2.(..(tn.Nil)..)) is the name of the list [t1,t2,..tn]. The set of all such terms is the recursive data structure implicitly introduced by this logic program. Note that the 'cases' treated by each clause are indicated by the 'patterns' of terms in the consequent atom of the clause, the 'procedure head' as Kowalski calls it. This is a common form of logic programs for relations on recursive data structures. Since we usually give the form of both the 'input' and 'output' structures, it corresponds to a slight generalisation of the

case format proposed by Hoare [1973] for programs manipulating recursive data structures.

Computational use

Suppose we want to append two lists, say the two unit lists [2] and [3]. We do this by asking for an instance of the unary relation:

x is the result of appending [2] and [3].

Under our assumed meaning of the "append" predicate, this relation is named by append(2.Nil,3.Nil,x). A request for an instance of this relation is written:

```
<-append(2.Nil,3.Nil,x).
```

Such an expression is a goal clause, the atom append(2.Nil,3.Nil,x) being a call of the append program. An evaluation of the goal clause is a constructive proof, using the program clauses as premises, that there is an x such that append(2.Nil,3.Nil,x).

The proof is constructive in that it will bind x to a term t such that the assertion append(2.Nil,3.Nil,t) is a logical consequence of the program clauses. Since these clauses are all true statements about appending lists, the assertion append(2.Nil,3.Nil,t) must be a true statement about the append relation. This means that x can only be bound to the term 2.3.Nil. This is the only term that names the list [2,3] which is the result of appending the lists [2],[3].

The truth preserving aspect of a logic program computation is its crucial property. We shall deal with this more fully in Chapter 3. For the time being let us note that it enables us to understand a logic program **declaratively** or **procedurally**; to understand it as a set of statements about the relation we want to compute, or as a recipe for finding instances of the relation.

The declarative reading of the "append" program is as a pair of universally quantified statements about appending lists. Its procedural reading depends on the way it is used. As a program for appending lists it can be read:

```
to append the empty list Nil to some list y, return y;
to append a constructed list u.x to some list y first
append x to y giving z, then return u.z.
```

Non-deterministic use

We can use the same logic program non-deterministically to split a list into front and back sublists. To split the list [2,3] we use the goal clause

```
<-append(x,y,2.3.Nil)
```

since this names the relation

{<x,y>: [2,3] is the concatenation of x,y}.

As we shall see, each of the substitutions

{x/Nil,y/2.3.Nil}

{x/2.Nil,y/3.Nil}

{x/2.?.Nil,y/Nil}

is a possible answer substitution. Each correctly denotes an instance of the relation named by the goal clause. The ability to use the same set of clauses to compute both a function and its inverse, and more generally to find values for any unknown arguments of a relation, is a novel feature of logic programs. Indeed, as we shall see in the next chapter, each logic program represents a whole family of different algorithms, each algorithm corresponding to a different computational use of the clauses of the program.

As a program for decomposing a list into front and back sublists, the "append" program can be read:

```
to decompose a list w
either return <Nil,w>
or     if w is a constructed list u.z then
           decompose z into <x,y> and return <u.x,y>
```

The **either**,**or** indicates a non-deterministic branch.

Most general answers

The goal clause

<-append(x,y,z)

is a request for any triple of lists in the append relation. There are an infinite number of instances of this relation. The goal clause evaluator we describe in section 1.4 is able to construct answer substitutions that 'cover' all these instances. More interestingly, each answer substitution names not just a single instance, but an infinite subset of the relation. One answer is

{x/Nil,y/y,z/y},

which, since it binds y and z to the same variable, denotes the infinite set of list tuples

{<[ ],l,l>:l any list}.

Another answer is

{x/u.Nil,y/y,z/u.y},

which names the infinite set of append instances that have a unit list as first argument. The general form of each of the infinite set of

possible answer substitutions is

{x/u1.u2...uk.Nil, y/y, z/u1.u2...uk.y}, k≥0.

It denotes the infinite set of instances that have a list of length k as first argument. We get these instance schemas, rather than particular instances, because the goal clause evaluator always returns the most general answer for each inference/computation path.

As a program for generating the general form of an instance of the append relation the program can be read:

```
either return <Nil,x,x>
or     find an instance <t1,t2,t3> and return <u.t1,t2,u.t3>
           where u is a variable not appearing in t1,t2 or t3.
```

example program-2

The generation and modification of answer substitutions are the data structure construction and manipulation operations of a logic program evaluation. Viewed as data structures, answer bindings that contain variables have a special role. This is because the answer binding x/t, where t contains some variable y, is modified to a new binding x/t' when the variable y is itself bound. This gives the effect of a data structure assignment (t changed to t').

One use of the following program manipulates answer bindings in just this way. It is a program for the predicate "front". This is intended to name the relation which holds between a number n and a pair of lists z,x when x is the list of the first n elements of z. It makes use of the above program for "append", and an auxiliary program for the list length relation.

```
front(n,x,z)<-length(x,n) & append(x,y,z)

length(Nil,0)<-
length(u.x,s(n))<-length(x,n)
```

The clauses can be read:

x is the list of the front n elements of z if the length of x is n and there is a y such that x appended to y is z,

the length of the empty list is zero,

the length of a list u.x is n+1 if the length of x is n.

These are all true statements about the front relation.

Let us note in passing that the program implicitly 'declares' two recursive data structures. The first is the list data structure, the one we have already come across. The other is the recursive data structure of the natural numbers, the set of objects generated from zero, named by "0", using the number generator, add 1, named by the functor "s". Thus, the term "s(0)" names the number 1, "s(s(0))" the number 2, and so on. Of course, we can, and shall, use the normal

decimal notation numerals as syntactic sugar for these terms.

Relative to the above reading of the program as a set of statements about the front n elements of a list, the goal clause <-front(2,x,A.B.C.Nil) names the unary relation that is true of x when it is the list [A,B] of the first two elements of the list [A,B,C]. One computational use of the program to generate the answer substitution {x/A.B.Nil} (we shall discover in the next chapter there are other quite different ways to generate the answer), is equivalent to first evaluating the goal clause <-length(2,x) to produce the answer substitution {x/u.v.Nil}, then evaluating the goal clause <-append(u.v.Nil,y,A.B.C.Nil) to produce the auxiliary bindings {u/A,v/B}. The first answer binding for x gives the general form of a list of two elements, the second answer binding is the final answer {x/A.B.Nil}. This is generated when the evaluation of <-append(u.v.Nil,y,A.B.C.Nil) 'fills-in' the u,v slots of the general answer.

example program-3

The two sets of assertions:

```
Jack fathered George<-        Tom  married Mary<-
Tom  fathered Bill<-          Bill married Jane<-
  .                             .
  .                             .                    (3)
  .                             .
Bob  fathered Tom<-           Jack married Susan<-
```

are a quite different kind of logic program. They are more like a data base than a program in the conventional sense. With the English language meaning of the relation names, we can read them as a description of certain family relations that prevail amongst a group of people named "Jack", "George", etc.

Data retrieval

We 'compute' with the data-base-style program in exactly the same way as with the 'recursive' program for append. Thus a goal clause

<-Tom fathered x & x married Jane

is a request for an instance of the unary relation (presumably the only instance) comprising offsprings of Tom that are married to Jane. The evaluation of this goal clause will be a search over the "fathered" and "married" assertion sets. The result is the binding x/Bill.

Data confirmation

The goal clause

<-Bob married Sarah

is a request for the confirmation that Bob and Sarah are married. If the assertion

Bob married Sarah <-

appears in our logic program the answer is **true**. If the assertion does not appear, all attempts to establish that Bob is married to Sarah using the assertions of the program will fail, so the answer is **fail**.

1.3 Substitution and unification

The abstract interpreter we shall describe in the next section is a resolution theorem prover. The unit of computation is a resolution, of which the essential component is unification. Unification is the process of finding a substitution that makes two or more atoms syntactically identical. In this section we give a brief introduction to these concepts. For a complete treatment the reader should consult Robinson[1965] or Robinson[1979].

DEFINITIONS
(1) A **substitution** is a set

$$s = \{x1/t1,..,xk/tk\}$$

of variable/term pairs in which the x1,..,xk are distinct variables. We say that xi is bound to the term ti.

(2) A **substitution instance** of an expression E (a clause, atom or term) is any expression E' that can be obtained from E by simultaneously replacing each of the variables bound by a substitution s, at each occurence in E, by the term to which it is bound. We use [E]s or Es to denote this substitution instance.

(3) A **variant** of an expression E is a substitution instance Es where s is a change of variable substitution. That is,

$$s = \{x1/y1,x2/y2,..,xk/yk\}$$

where x1,..xk are all the variables of E and y1,..,yk are any k distinct variables.

(4) The **composition** s1*s2 of two substitutions

$$s1 = \{x1/t1,..,xn/tn\},\quad s2 = \{y1/t'1,..,yk/t'k\}$$

is the substitution

$$s'1 \text{ union } s'2$$

where

$$s'1 = \{x1/[t1]s2,...,xn/[tn]s2\}$$

and s'2 is s2 with any bindings for the variables x1,..,xn deleted.

examples

C is the clause

P(f(x,a),g(b,y)) <- Q(x,z) & P(y,z)

and s1 = {x/u,y/v,z/x}
s2 = {x/h(u),y/x,z/b}
s3 = {u/g(b),x/h(u)}.

[C]s1 is the C variant

P(f(u,a),g(b,v)) <- Q(u,x) & P(v,x).

[C]s2 is the substitution instance

P(f(h(u),a),g(b,x)) <- Q(h(u),b) & P(x,b).

s2°s3 is the substitution

{x/h(g(b)) , y/h(u) , z/b , u/g(b)}.

Notes
(1) An expression variant just 'says the same thing' using different variables. Two expressions E,E' are instances of one another if and only if they are variants.

(2) The notation and definition of a substitution is a little different from the standard one. Normally a binding of a term t to a variable x is written t/x. We use x/t because it accords with the variable assignment notation x:=t of programming languages. The definition is a little different because the standard definition excludes identity bindings such as x/x. They are excluded because they have no effect when the substitution is applied. We allow identity bindings in substitutions since we want to include them in the answer substitutions produced by a logic program evaluation. The consequence is that different sets of bindings s1 s2 can represent the same substitution. Our definition of equality for substitutions is:

s1=s2 iff s1 and s2 are identical sets of bindings
after the deletion of any identity bindings.

Thus the identity substitution, the substitution that leaves any clause unchanged, is denoted by any set of identity bindings, including the empty set { }.

(3) Composition of substitutions is defined so that

[Cs]s'=[C]s°s'.

It also has the property that composition is associative. In consequence we shall leave out the brackets in expressions such as

s1°s2°s3.

DEFINITION
A **unifier** of two atoms A1,A2 is a substitution s such that [A1]s and [A2]s are syntactically identical. s is a **most general unifier** (m.g.u) if every other unifying substitution s' is such that s'=s°s" for some substitution s".

Example

The substitution

{w/2.z,u/2,x/Nil,y/3.Nil}

is a unifier for the pair of atoms

append(2.Nil,3.Nil,w), append(u.x,y,u.z).

When applied to each atom it produces the same substitution instance

append(2.Nil,3.Nil,2.z).

Any other unifier must have the same bindings for u,x and y but could bind w to any term of the form 2.t if it included the binding z/t. Since such a substitution is the composition of the above unifier and the substitution {z/t}, the above unifier is an m.g.u. of the atoms.

Unification algorithm

An algorithm which tests whether or not two atoms are unifiable, and which returns an m.g.u. if they are, is a unification algorithm.

In the logic programming context the unification of two atoms is the 'data handshake' between a goal clause atom and the consequent atom of some program clause that takes place when the evaluator makes use of the clause. In the above example, append(2.Nil,3.Nil,w) would be the goal clause atom and append(u.x,y,u.z) the consequent atom. Unification does the work of data structure component selection (the bindings u/2, x/Nil), and unification and composition does the work of data structure construction (the binding w/2.z composed with some binding for z).

Robinson[1965] gave the first unification algorithm and proved it correct. Since then there have been several faster algorithms proposed, for example in [Baxter 1973] and in [Patterson & Wegman 1976]. Unfortunately these faster algorithms require a special representation of the atoms to be unified. Robinson's algorithm will work directly on the list/sub-list representation of the atoms used in logic programming implementations. Consequently his original algorithm (or rather, as we shall see below, a much faster restricted version) is used by the PROLOG systems.

Robinson's algorithm

Suppose the atoms to be unified are R(t1,..,tn) and R(t'1,..,t'n). The algorithm iterates down the list of argument pairs [<t1,t'1>,..,<tn,t'n>] building up a sequence s1,s2,..,sk of partial unifiers. If the algorithm successfully terminates, the composition s1°s2°..°sk is an m.g.u. of the two atoms. At the beginning, the sequence of partial unifiers has just one substitution, which is the identity substitution.

When the list of argument pairs is empty the algorithm successfully terminates.

Otherwise, suppose <t,t'> is the first argument pair on the list. We must try to unify ts and t's where s is the composition s1°s2°..°si of the sequence of partial unifiers generated so far.

Case (1) ts or t's is a variable v:

```
if the other term is a non-variable e in which v
   occurs (the occur check), fail
else add {v/e} to the end of the sequence of partial
   unifiers and continue with the tail list of argument pairs.
```

Case (2) ts is a constant, t's is not a variable:

```
if t's is an identical constant continue with the tail list of
   argument pairs,
else fail.
```

Case (3) ts is of the form f(e1,..,em), t's is not a variable:

```
if t's is of the form f(e'1,..,e'm) continue with the list
   [<e1,e'1>,..,<em,e'm>] appended to the front of the
   tail list of argument pairs,
else fail.
```

A note on implementation

In applying the above algorithm we do not need to construct the terms ts and t's. We can instead use t and t'. However, when we encounter a variable we must first check to see if it is bound by one of the partial unifiers s1,..,si. If it is, we proceed as if its binding term appeared instead of the variable. In effect, we are modifying the algorithm so that t,t' and the set of previously generated bindings represent ts and t's. By storing this set of bindings in such a way that we can rapidly check whether a variable is bound, and if so, to what binding term, we can considerably speed-up the execution of the algorithm. We shall say more about this implicit repesentation of substitution instances, and the speedy look-up of variable bindings, in section 1.6.

The occur check

The slowness of the above algorithm is entirely due to the occur check. A simple example of an occur check failure is the pair of atoms

P(x), P(f(x)).

The attempted unification will pair x with the term f(x) in which it appears. In this case it is obvious that the atoms are not unifiable. Any substitution {x/t} will produce the non-identical pair of atoms P(t), P(f(t)). A more interesting example of an occur check failure is the pair of atoms

P(x,g(x)), P(h(y),y)

that have no variables in common. The pairing of x with h(y) will generate the partial unifier {x/h(y)} which now modifies the pair <g(x),y> making them the argument pair <g(h(y)),y>. The match of y and g(h(y)) gives an occur check failure.

Fortunately, most of the pairs of atoms that must be unified during a logic program evaluation satisfy two condititons that exclude the possibility of an occur check failure. The atoms never have variables in common, and the goal clause atom usually only has a single occurrence of each variable. For a pair of atoms satisfying these two conditions, an occur check failure cannot arise. Because of this, none of the PROLOG implementations apply the occur check, although in IC-PROLOG it can be requested for unifications involving specified clauses. The gain is extremely fast unification. The loss (except for IC-PROLOG) is that the logic programmer must make sure that his program is such that an occur check failure would not arise.

1.4 An abstract interpreter

Any resolution theorem prover can be used as a logic program executor. However the inference system which is most obviously computational is LUSH resolution [Hill 1974]. This is the inference system that Kowalski describes when he talks about the procedural interpretation of predicate logic [Kowalski 1974].

The LUSH inference rule defines a search space of alternative derivations. We prefer to think of these alternative derivations as alternative paths of a non-deterministic evaluation of a 'call' of the logic program given by some **goal clause**

<-B1&..&Bn.

Here, B1,..,Bn are all atoms, and, if x1,..,xk are the variables of the clause, it is a 'call' for the computation of a substitution

s = {x1/e1,x2/e2,..,xk/ek}

such that [B1&..&Bn]s is a logical consequence of the program. If there are no variables in the goal clause, it is request for the confirmation that B1&...&Bn is a logical consequence of the program.

The trace of one of the paths of the non-deterministic evaluation is given by a sequence of goal clauses

C1,C2,...,Cn,....

each one derived from the preceding one, by the following evaluation step. C1 is the initial, or given, goal clause. We shall follow Kowalski [1974] and refer to the program clauses as procedures, the consequent atoms of the clauses as procedure heads, and the antecedent atoms as procedure calls. We shall also refer to the atoms of a goal clause as procedure calls.

Computation rule

Suppose the current goal clause comprises the conjunction of procedure calls

<-B1&...&Bk.

An evaluation step is the execution of one of these calls. Which one is executed is determined by a **computation rule**, a rule which for goal clause generated during the evaluation selects one atom of the clause as the call to be executed next. The computation rule is a parameter of the evaluation.

Evaluation step

Let us suppose that the selected call is Bi, and that this is the atom R(t1,..,tn). An attempt is made to start the execution of the call using a variant

R(t'1,..,t'n)<-A1&...&Am

of one of the program procedures for R. This variant must have no variables in common with the current goal clause. If there is more than one procedure for R, the choosing of the procedure is a non-deterministic step in the evaluation, each procedure about R presenting us with an alternative branching of the evaluation.

We try to unify R(t1,..,tn) with R(t'1,..,t'n).

If they do not unify this branching of the evaluation path terminates with **fail**.

If they do unify, with most general unifier s, this branching of the evaluation path leads to a new conjuction of procedure calls, a new goal clause, which is the resolvent

<- [B1&..&Bi-1& A1&..&Am &Bi+1&..&Bk]s.

Coroutining

If, for the next evaluation step, the computation rule selects one of the newly introduced calls [A1]s,..,[Am]s, the evaluation is continuing with the execution of the previously selected call. A computation rule that always selects a most recently introduced call corresponds to the call, execute, return control of a conventional programming language. If the computation rule does not select one of the calls [A1]s,..,[Am]s, it is suspending the execution of the Bi call, and starting the execution of one of the other calls [Bj]s. We shall call such a rule a coroutining rule. We shall view any binding in the substitution s for a variable that appears in Bj as 'data' that is communicated to Bj. In the next chapter we shall discuss the role of the computation rule in more detail. In particular, we shall discuss data flow computation rules which coroutine on the basis of the communication of data through shared variables.

Successful evaluation

An evaluation path terminates with **success** when the empty goal clause is generated. This happens when the last step is the execution of the single call of a goal clause <-B using an assertion procedure B'<-.

Computed answer substitution

Suppose s1,...,sn is the sequence of unifying substitutions of a successful evaluation, s1 being the substitution of the first step, sn that of the last step. Let

s = s1°s2°...°sn

be the composition of these unifying substitutions. The subset of s that gives bindings for the variables of the initial goal clause C, augmented with the identity substitution x/x for any variable of C not bound by s, is the answer given by that successful evaluation. If there are no variables in C, the answer is **true**.

DEFINITION
An answer substitution for a goal clause C and program P is R-computable if it is the output of a successful evaluation of C,P using computation rule R.

Procedural vs. non-procedural semantics

The above definition is our procedural semantics for answer substitutions. It characterises answer substitutions in terms of a mechanism for computing them. In Chapter 3 we shall give a non-procedural semantics for answer substitutions using the model theory of first order logic. We shall prove that this model theory semantics defines essentially the same set of answers as the procedural semantics. This is what justifies the dual declarative/procedural reading for logic programs. We shall also prove that the set of computable answers is independent of the computation rule.

1.5 Search trees

A logic program P, a goal clause C, and a computation rule R, together define a search tree of all the possible evaluation paths for C. The form of the tree is as depicted in Fig. 1.1. Each interior node in the tree is a clause derived from its parent by an evaluation step. Each of the children of a node are the results of attempted executions of the selected procedure call of that node. A fail node offspring records a

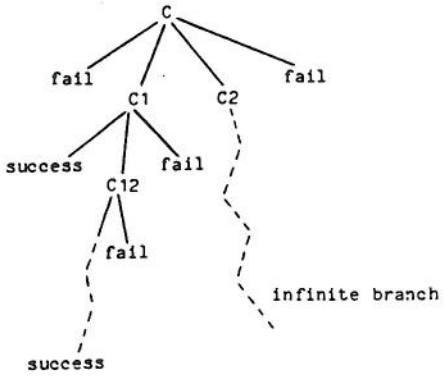
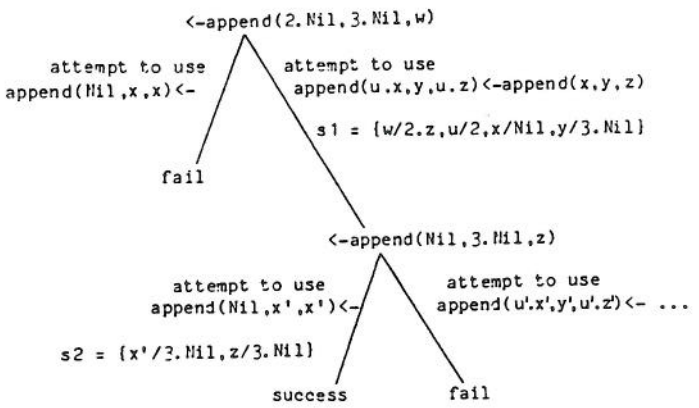


Fig. 1.1 A search tree

failed unification. A success leaf node marks the end of a successful evaluation path. Some of the evaluation paths may be infinite.

Figs. 1.2 and 1.3 are search trees for two of the calls to the logic programs of the preceding section. The computation rule used is: select the leftmost call.

The very first step of the succesful evaluation path of Fig. 1.2 requires the unification of append(2.Nil,3.Nil,w) and append(u.x,y,u.z). The m.g.u. binds u to 2 and x to Nil, that is it decomposes the 'input' 2.Nil into its head and tail sublists. As we mentioned in the preceding section, the selection of components from data structures in accordance with a given pattern, in this case the pattern "u.x", is one of the major roles of unification. The other major role is the production of partial approximations, templates, for the output. This is illustrated by the binding 2.z for w. This binding gives us a first approximation to the answer substitution w/2.3.Nil which is the final output of the evaluation path. It tells us that any answer for the path will bind w to 2."something". This first approximation is a partial result that could now be accessed by another procedure call in which w appeared. We shall return to this idea of 'data flow' activation of procedure calls in the next chapter.



Answer: s1•s2 binds w to 2.3.Nil

Fig. 1.2

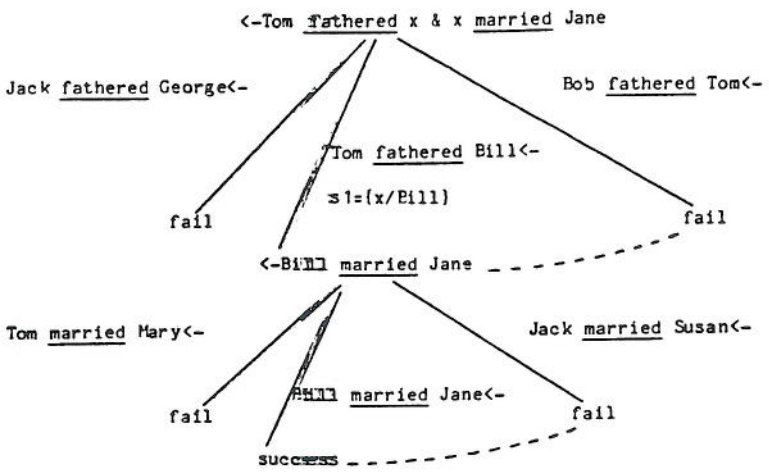
Search rule

The strategy that is used to search for a successful evaluation, the search tree construction strategy, we shall call the **search rule**.

The search rule must be fair. That is, it must be such that each success branch on the search tree will eventually be found. For search trees on which each branch is finite (and so, by König's lemma contain only a finite number of nodes) a back-tracking search as depicted in Fig. 1.4 is fair. For a back-tracking search strategy the seach rule reduces to an ordering rule, a rule which for each selected atom gives the try order of the program clauses for its predicate. The ordering rule defines the left to right order of the branches on the search tree.

A back-tracking search for the successful evaluation of Fig. 1.3 is equivalent to a double loop search over the "fathered" and "married" assertion sets. However, a similar depth-first search of Fig. 1.2 does not involve significant back-tracking. Each failure node is an immediate offspring of the one and only successful evaluation path. Search trees such as this record an essentially deterministic computation. At each step there is only one clause whose consequent will unify with the selected call of the goal clause.

A genuinely non-deterministic use of the append program is depicted in Fig. 1.5. It is the search tree for the goal clause <-append(x,y,2.3.Nil). Each of the answer substitutions gives one of the possible decompositions of the list [2,3] into front and back sublists. A back-tracking construction of this tree will generate the answer substitutions in the left-to-right order that they appear on the



Answer: leftmost successful evaluation binds x to Bill

Fig. 3.1

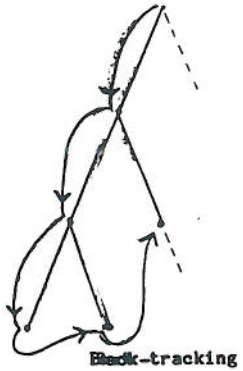


Fig. 1.4

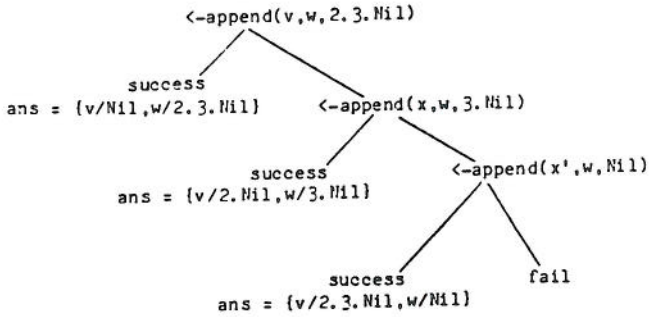


Fig. 1.5

tree.

1.6 Proof trees and spaghetti stacks

We have defined the evaluation step of the abstract interpreter as a textual substitution in a goal clause 'state of evaluation'. In an implementation we would, of course, represent this substitution implicitly. We do not want to go into too much detail concerning implementation. However, we can get the general idea of how the interpreter can be implemented using activation records and binding environments, as well as an alternative conceptualisation of what an evaluation is, by viewing a successful evaluation as the construction of a **complete proof tree.**

DEFINITIONS

For a goal clause C and logic program P:

(1) A **proof tree** is a tree whose non-root nodes are all labelled by substitution instances of atoms in C and P in accordance with the following constraints:
(a) the (unlabelled) root node has n immediate descendants labelled with atoms B'1,..,B'n only if <-B'1&..&B'n is a substitution instance of the goal clause. We call these the goal clause atoms of the tree.
(b) a node labelled B' has m immediate descendants labelled A'1,..,A'm only if B'<-A'1&..&A'm is a substitution instance of some clause in the program.

(2) The **initial proof tree** is a proof tree whose leaf atoms are all immediate descendants of the root node and these are labelled by the atoms of the goal clause.

(3) A **terminal node** is a leaf node of a proof tree labelled by a substitution instance B' of a program assertion.

(4) A **complete proof tree** is a proof tree whose leaf nodes are all terminal nodes.

A successful evaluation can now be re-interpreted as the construction of a complete proof tree. The construction starts with the initial proof tree for the goal clause and program and proceeds by a sequence of proof tree extension operations. The extension operation is the proof tree equivalent of a goal clause evaluation step. At each stage in the construction the leaf atoms not yet marked as terminal are the atoms of the derived goal clause.

Proof tree extension

Using the computation rule, we select one of the unmarked leaf atoms atoms Bi. If Bi unifies with the consequent atom of some clause variant B'<-A'1&..&A'm, which does not contain any variable already on the tree, we modify the tree as follows:

(1) If m=0, mark the selected leaf as terminal.
If m>0, add A1,..,Am as immediate descendants of the selected leaf.

(2) Apply the unifying substitution to each atom of the tree.

DEFINITION
A **constructed proof tree** is a proof tree that has been constructed from the initial proof tree by a sequence of proof tree extensions.

Constructed proof trees are the abstract data structures manipulated by the stack implementation we shall shortly describe. Proof trees, constructed or not, are the crucial intermediary concept which will enable us to connect the procedural semantics for answer substitutions with the non-procedural semantics of Chapter 3.

Structured shared representation

When we extend a constructed proof tree we do not need to literally apply the unifying sustitution to each atom on the tree. We can instead 'associate' the set of variable bindings that represent the unifier with the tree. Then, when we are attempting to unify a selected leaf atom with some procedure head, if we encouter a variable we not only check to see it has already been bound during the current unification, we also check to see if it was bound by any of the previous unifiers. In other words, we 'evaluate' each atom relative to all the bindings so far generated.

Fig.1.6 is an example of a constructed proof tree represented in this implicit fashion. It is for a goal clause <-B1&B2&B3 and a program which contains:

a clause B<-A with a variant [B<-A]v such that Bv unifies with B1 with m.g.u. s1,

a clause B'<-A1&A2 with a variant [B<-A1&A2]v' such that [B']v' unifies with [B3]s1 with m.g.u. s2,

an assertion A'<- with a variant [A'<-]v" such that

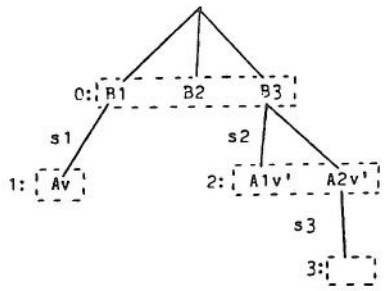


Fig. 1.6

[A']v" unifies with [A2v']s1*s2 with m.g.u. s3.

The numbers attached to each sibling group of atoms give the extension step at which that group of atoms was added to the tree. The attaching of the empty sibling group as the immediate descendant of the atom A2v' is the marking of [A2v']s1*s2 as a terminal node.

Implicit representation of atoms and binding terms

In a tree as depicted in Fig. 1.6 the i'th sibling group of atoms are always the antecedent atoms of the clause variant used in the i'th extension step. If we generate the variant by subscripting the variables of the clause with the subscript i, then
(1) we shall always get a variant that does not contain variables already on the tree, and
(2) the sibling group of atoms can be represented by a pointer to the program clause.

The pointer to the clause and the step label i determine the variants of its antecedent atoms that should appear on the tree. For example, in Fig. 1.6, a pointer to the program clause B'<-A1&A2, together with the step number 2, would identify the atoms A1v', A2v' as A1 and A2 with their variables subscripted by 2.

The same implicit representation can be used for variable bindings. Suppose that in the construction of the above tree, the second step unification bound some varable x0 of the goal clause to a term [t]v of the atom B'v. This is just the term t of B' with its variables subscripted by 2. We can therefore represent it by the term/step number pair <t,2>, storing the x0 binding as x0/<t,2>. As with the program clauses, t will be represented by a pointer to its occurence in B'.

This implicit representation of the atoms and the binding terms is the Boyer and Moore[1973] structure shared representation of resolvents.

Binding environments

Remember that when we extend the tree we must 'evaluate' the selected leaf relative to the sets of bindings of the previous unifiers. We do this by looking-up any variable we encounter in each of the previous unifiers. If it is bound by any of them, and it can be bound by at most one, then we proceed as if its binding term appeared in place of the variable.

For speedy look-up of the variable bindings we can re-distribute them over the tree. We can group all the bindings for an i-subscripted variable with the i'th sibling group of nodes on the tree. Looking-up an i subscripted variable is now a matter of checking whether that variable is bound in just the subset of substitutions associated with the i'th sibling group. By 'compiling' variables of a program clause into integer offsets, and assigning a binding location for each variable whether or not it is bound, looking-up the j'th variable of the clause variant used in the i'th extension step becomes an access-operation on the j'th location of the i'th vector of bindings. Conversely, recording the binding when a unification step binds the variable is a store operation on this location. That a variable is not yet bound, can be recorded by storing a special "undefined" value in its binding location.

Our constructed proof tree of Fig. 1.6 is now a tree of activation records as depicted in Fig.1.7. Each activation record points to a program clause, and contains a vector of binding locations which is the binding environment for that activation of the clause. The ei are the binding environments.

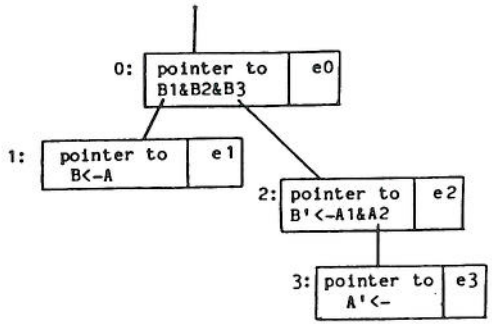


Fig. 1.7

Representation as a stack

Fig. 1.8 is the linear representation of Fig. 1.7 as a spaghetti stack. Each entry on the stack is an activation record for a clause.

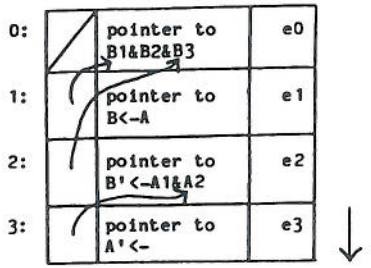


Fig. 1.8

When we extend the proof tree we grow the stack. The unmarked leaf atoms of the tree are the atoms of an activation record which are not pointed to by an activation record lower down the stack. To save checking this, we can include a live-dead bit vector in each activation record. As soon as the j'th atom of an activation record is pointed to, we set the j'th bit.

Popping on failure

Such a stack representation is ideal for implementing a back-tracking search. Let us suppose that in the proof tree depicted in Fig. 1.6, and represented by the stack of Fig. 1.8, the computation rule selects the atom [A1v']s1^{a}s2^{a}s3. This is <A1,2> evaluated relative to all the defined bindings of e0,e1,e2 and e3. Let us further suppose that this atom will not unify with any of the program clauses for its predicate. As we try each clause in turn, we are back-tracking over all the failure nodes which are the immediate descendants of

[(Av)&B2&(A1v')] s1^{a}s2^{a}s3

on the partially constructed search tree of Fig. 1.9. A back-tracking traversal of the search tree would next visit the descendant of

[(Av)&B2&(A1v'&A2v')] s1^{a}s2

given by trying to unify [A2v']s1^{a}s2 using an untried clause for its predicate. To jump to this clause on the search tree, we must first reset the stack so that it records the derived clause [(Av)&B1&(A1v'&A2v')]s1^{a}s2. We do this by popping the top record of the stack, and deleting (in fact setting to undefined) any bindings introduced into the binding environments e0,e1,e2,e3 by the unifying substitution s3. To know which variables to reset, we must include with each stack entry a reset list of those variables in binding environments higher up the stack bound by the evaluation step it records. The only

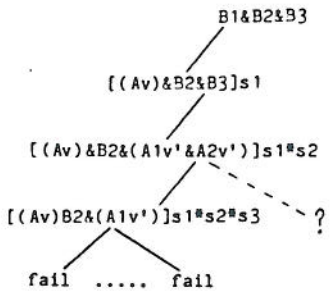


Fig. 1.9

other information we need is the next clause to be tried for the alternative evaluation step involving [A2v']s1*s2. So we must also have recorded in the popped activation record (implicitly or explicitly) the list of untried clauses for its parent atom. If this set is empty, we backtrack further by again failure popping the stack.

Popping on success

In the stack implementation of a more conventional recursive programming language a top of stack activation record with no unexecuted procedure calls would be popped from the stack. Such an activation record records a successfully completed evaluation of its parent procedure call. The top of stack record on the stack of Fig. 1.8 is an example of such an activation record. Its removal would be a success popping of the stack.

For us, success popping is not so straightforward. To begin with the activation record should not be removed if it records an evaluation step for which there are untried alternatives. If this is the case, its reset list and list of untried clauses will be needed if we subsequently have to back-track on that evaluation step and failure pop the record. Its entry on the stack records a branch point in the evaluation which must be remembered. It might as well be remembered by leaving the record on the stack.

Even when the top of stack activation record has an empty list of untried clauses, we cannot simply discard it. We must first append its reset list of variables to the reset list of the next to top activation record. Failure popping this next to top record is now equivalent to failure popping the top two records one after the other. We must also do something with its 'referenced' variables. These are the variables of the top of stack binding environment ei that appear in the binding terms of variables in e0,..,ei-1. The bindings for these 'referenced' variables are part of the substitution that is implicitly applied to each unactivated procedure call on the stack. The bindings for the refenced variables must be retained.

One solution, that adopted by Warren[1977], is to keep the bindings for the variables that can be referenced in this way on a separate 'global variable' stack. This stack can be failure popped but not success popped. The variable bindings that are kept in the activation stack are for the 'local' variables, the variables of the clause that cannot appear in the binding terms of variables higher up the stack. The bindings for the local variables can be garbage collected by a success pop.

Tail recursion

When we success pop the stack we make sure that the reduced stack retains all the information about the state of the evaluation that we may need for a subsequent evaluation step, or a subsequent back-tracking step. Using the same 'compression' techniques, we can sometimes record the state of the evaluation after an evaluation step without growing the stack.

Let us suppose that, as for a success pop, the current top of stack record records a deterministic evaluation step, a step for which there are no untried alternatives. Let us further suppose that it has one unexecuted call which is the selected call for the next evaluation step. If the execution of this call is also a deterministic step, then we can success pop the top of stack record before we grow the stack.

This space optimisation is a generalisation of what is usually called tail recursion. In the special case where the new activation record is for the same program clause, the overwriting of the variable bindings is equivalent to a sequence of assignments to the local variables of the clause for an 'iterative' execution. The Edinburgh PROLOG anticipates this possibility, and compiles programs such as the "append" program into iterative code [Warren 1979].

The above has been an intentionally brief introduction to the implementation possibilities. For further details the reader should consult the referenced papers by Warren.

Chapter 2 Logic Algorithms

For the abstract interpreter we defined in the last chapter, the computation rule and the search rule are a 'control input' which is independent of the logic program and goal clause. By changing this control input we change the way the answer substitutions are computed, but, providing the search rule is such that it will eventually find each successful evaluation, we do not change the set of computed answers[1].

Following Hayes[1973] and Kowalski[1979a], we can think of each computation of this set of answers as the trace of one of the family of equivalent algorithms implicit in the logic program, each algorithm in the family being defined by the program and a particular control input to the interpreter.

DEFINITION
A **logic algorithm** is a logic program together with a fixed computation and search rule control.

In this chapter we shall discuss some search and computation rule controls that are both 'algorithmically' useful and straightforward to implement. As a focus for the discussion, we shall examine the search and computation rule control options available to the PROLOG programmer, and in particular, the IC-PROLOG programmer. We shall describe the way in which the control that is to be used in conjunction with a particular PROLOG program is specified, and we shall look at the sort of algorithm transformations that can be brought about just by changing this control.

2.1 Search rule control

We have seen that back-tracking is very easy to implement. For this reason all the PROLOG implementations use back-tracking as the search strategy. This means that the programmer control of the search is limited to the ordering rule, the rule which determines the try order of the program clauses for each selected call. Although back-tracking is not, in general, a fair strategy, this restriction of the search rule can nearly always be compensated for. By a judicious choice of the computation rule and the ordering rule we can generally ensure that a back-tracking search would find all the successful evaluations.

The ordering rule could be such that it assigns a different try order for different calls involving the same predicate. The implementation of such a context dependent ordering rule need not impose too much overhead. MICRO-PLANNER[Sussman & Winograd 1970], with its recommendation lists, provided a restricted form of dynamic ordering.

---

1 This is because the set of R-computable answers is independent of the computation rule R. A result we have already mentioned, which we shall prove in the next chapter.

As yet, no PROLOG implementation gives this flexibility. The programmer must choose a fixed try order for each call involving the predicate. He then implicitly specifies this try order by the before-after order in which he enters the clauses.

Clause indexing

There is one refinement of the fixed try order that several of the PROLOGs allow. This is the indexing of the clauses. By accessing the clauses for a particular predicate via an index, we can often delete from the try list some or all of the clauses that cannot unify with the call. This cuts out much of the shallow back-tracking of the search for an applicable clause.

The Edinburgh PROLOG automatically indexes the clauses for every predicate on the top-level functor or constant of the first argument position. In IC-PROLOG, indexing for a predicate must be explicitly requested, but the indexing can be on any or all the argument positions. Such indexing is particularly useful for data base programs. We shall see that by indexing we can significantly modify the retrieval algorithm of some logic + control combination.

2.2 Computation rule control

The computation rule is an area in which we can more usefully exploit control flexibilty. In this, IC-PROLOG differs significantly from the Marseilles and Edinburgh PROLOG implementations. Both of these use a depth first left-to-right proof tree construction strategy. The programmer just controls the left to right order of each sibling group of nodes on the tree. Each of these sibling groups comprise some instance of the procedure calls of a program clause. By the left to right order in which these calls appear in the clause, the programmer fixes their order on the proof tree for each use of the clause.

The depth first construction of the proof tree corresponds to the strictly sequential execution of a conventional program. When a program procedure

B<-A1&..&Am

is used, 'control' will 'enter' the procedure and 'exit' only when each of its procedure calls A1,..,Am have been executed, one after the other, in the order in which they are given. The execution of each call is the construction of the sub-tree of the complete proof tree that is rooted at that call.

The major advantage of using such a simple computation rule is that it is easy for the programmer to understand and very easy to implement. Unfortunately, it does somewhat restrict the range of algorithms that we can obtain from a given logic program.

We have seen that a logic program can be called with different input/output patterns, with different arguments given and different arguments to be found. For each call pattern we may need a quite different order of execution of the calls in the body of each clause.

We should therefore allow the execution order to be determined by the use.

We have also seen that the execution of a call can, after a few steps, produce a partial result for the final output binding of a variable. For example, the execution of the call append(2.3.4.Nil,5.Nil,z), using our append program, will produce a whole sequence

2.w1, 2.3.w2, 2.3.4.w3, 2.3.4.5.Nil

of partial results ending in the final output binding z/2.3.4.5.Nil. The first partial result is produced when the first step in the execution binds z to 2.w1. Thereafter, each new partial result is generated when the next step in the execution binds the variable of the previous result. When a variable has its final output binding approximated to in this way, we could use the production of each new partial result as a trigger that suspends the execution of the call which is producing the binding, and resumes the execution of some call that is consuming it. The consumer call will be a call in which the variable also appears, but which is not allowed to generate a new partial result. The attempt to do this, by trying to bind some variable in the last partial result, would be the trigger for the suspension of the consumer and the resumption of the producer. This sort of data flow corouting is almost as straightforward a control concept as strictly sequential execution. It can also be implemented without too much overhead.

IC-PROLOG provides the programmer with this sort of control flexibility. The medium for specifying the computation rule is program annotation. As with the other PROLOGs, the text order of the procedure calls also carries implicit control information. A completely unannotated procedure will always have its calls executed in the order they are given, with the execution of the next call commenced only when that of the preceding call is complete. However, by attaching simple annotations to the variables of a procedure the programmer can:

(1) signal that the execution of a procedure call should be data flow coroutined with the execution of the sequence of preceding calls,

(2) restrict the use of the control defined by the text order and annotation of the procedure body to a particular input/output use.

This use of annotations to specify the computation rule control for Horn clause programs is similar to Schwarz's use of annotations to specify control information for recursion equation programs[Schwarz 1977].

Data flow coroutining

Suppose that in the clause

B<-A1&..&Am

the variable v appears in one or more of the calls A1,...,Ai-1 and in the call Ai. Let us also suppose that the execution of the procedure will

generate a sequence of term approximations to the final answer binding for v. The programmer signals that the execution of the sequence of calls A1&..&Ai-1 is to be data flow coroutined with the execution of Ai, by attaching a "^" or "?" annotation to the occurence of v in Ai.

Ai as the consumer of v

If v is annotated "v?", then Ai is the consumer and A1&..&Ai-1 the producer of the sequence of partial results for v.

Let us suppose that during an uninterrupted execution of A1&..&Ai-1 the final answer binding for v is approximated to by a sequence t1,t2,...,tn of partial results. The first partial result, t1, is generated when v is first given a non-variable binding. Thereafter, tj+1 is the term tj with one or more of its variables replaced by a non-variable term. It is generated when some some step in the evaluation of A1&..&Ai-1 binds one or more of the variables of tj.

The coroutining interaction between A1&..&Ai-1 and Ai starts with the execution of A1&..&Ai-1. However, the generation of each new partial result tj is the trigger for the suspension of the execution of A1&..&Ai-1 and the resumption (or start) of the execution of Ai. Conversely, the attempt by Ai to generate a new partial result for the binding of v, which it does when it tries to bind one of the variables of tj to a non-variable, is the trigger for the suspension of Ai and the resumption of A1&..&Ai-1.

Because Ai gets more 'information' about the final binding for v with each new partial result, we shall call the generation of each partial result a data transfer from the producer A1&..&Ai-1 to the consumer Ai. Correspondingly, we shall call the attempt by Ai to access more information than has been transfered, which it does when it tries to bind a variable of the last partial result, a data request from the consumer to the producer.

The data flow coroutining terminates when either the execution of Ai or that of A1&..&Ai-1 is completed. At that point, the currently suspended execution is resumed and run to completion.

Ai as the producer

If the variable in Ai is annotated "v^", then Ai is the producer of the sequence of partial results. The coroutining still starts with the execution of A1&..&Ai-1, but its execution is suspended as soon as it tries to bind v to a non-variable. Thereafter, the coroutining continues as for Ai the consumer, with the suspension and resumption triggers reversed.

A note on implementation

The above type of producer/consumer interaction can be implemented by using process descriptor bindings for variables. Each variable of a term that is the current partial result for the shared variable v is bound to a process descriptor. The process descriptor records the state of the consumer/producer interaction. It tells us whether the consumer or the producer is currently active, and it records the last suspension

point of each process.

We deal with these process descriptor bindings by modifying the unification algorithm. Suppose that, during some unification, there is an attempt to match a process descriptor binding for a variable u against a non-variable t. u must be a variable of the last partial result tj communicated from the producer to the consumer of the process descriptor. The match is the trigger for a control transfer.

If the process descriptor records a currently active producer, u is rebound to t, and all the variables of t inherit the process descriptor binding. This term binding for u has generated a new partial result tj+1. When the unification is successfully completed, control transfers to the suspended consumer. At the same time we update the process descriptor to record the new suspension address for the producer.

If the process descriptor records an active consumer, the unification is not completed. Any bindings it has already generated are undone. The consumer suspension address is updated, and the producer process resumed.

In each case the old state of the descriptor is saved in case we need to back-track om this step.

Inner coroutining

In a consumer/producer interaction between the executions of the call sequence A1,..,Ai-1 and the call Ai there can be inner coroutining.

In either execution a procedure may be invoked which has a local consumer/producer interaction signalled by an annotation on one of its shared variables. This inner coroutining will proceed independently of the outer coroutining between A1&..&Ai-1 and Ai.

Inner coroutining will also occur if one of the calls A1,..,Ai-1, say Aj, has been designated the producer or consumer of some other variable w[1]. If so, during the execution of the sequence A1&..&Ai-1 the execution of the call Aj will be coroutined with the execution of the subsequence of calls A1&..&Aj-1 in exactly the same way as the execution of A1&..&Aj-1&Aj&..&Ai-1 is coroutined wth Ai.

Lazy evaluation mode

The producer annotations of the procedure

R(x,w)<-T(w,z)&Q(z^,y)&R(y^,x)

will result in the generation of a sequence of partial results for w given x. This is exactly the behavior we want if the call is producing

1 In IC-PROLOG there can be only one designated producer or one consumer for any variable. The restriction to one producer is not unreasonable. The restriction to one consumer could be relaxed, with the consumers queued as described below.

w and consuming x. The Q and R calls are invoked 'lazily' to generate piecemeal results for the intermediary values z and y. By a consistent use of the producer annotation "^", we can specify a computational behaviour similar to that of lazy LISP [Henderson & Morris 1976, Friedman & Wise 1976].

Queuing of data transfers

When there is inner coroutining, a situation can arise in which several suspended executions should be resumed as a result of variable being bound. An example of this is the procedure

P(x,y)<-Q(x,y)&T(y?)

invoked by a producer P(x',y'^) to generate an answer binding for y'. When the first partial result is generated by the execution of the Q(x,y) call, we should both commence the execution of the consumer T(y?) and resume the execution of the consumer of y'. Intuitively, T(y?) should have precedence, since it is a local test of the partial result, which it may reject. This is what happens. The resumption of the innermost coroutined pair takes precedence.

The consumers that should be resumed when some variable is bound are queued in their inner to outer order. In turn, each is resumed as though it were the only suspended consumer. When the consumer generates the data request that normally leads to a resumption of the producer, the next consumer on the queue is resumed. Only when the queue is empty, is the producer re-activated.

In the above example, the test T(y?) will be executed first to check the partial result. If the check does not fail, the consumer of y' is resumed. On the next data request from this outer consumer, the same generate, test and transfer pattern will be repeated.

Delaying the data transfer

Normally, a producer is suspended immediately after the generation of the new partial result. By using another annotation, we can delay this producer suspension.

This is the clause bar. It is a ":" which replaces the "&" at any point in the sequence of calls of some procedure body, or which appears at the end of the sequence of calls. The effect of the bar in the clause

B<-A1&..&Aj:Aj+1&..&Am

is to delay the suspension of any producer that would normally occur during the execution of the procedure until after the execution of the sequence of calls A1&..&Aj is completed.

The bar can be used to delay the transfer of a new partial result until after some test procedure has successfully terminated. For

example, we could use

```
P(x,g(u))<-T(x):...
```

to delay the transfer of the partial result represented by the term g(u). It will only be transfered if the test on the x input succeeds. If this test fails, the incorrect partial result is never transferred.

It can also be used to delay the transfer of a new partial result until it is more fully instantiated. By using the bar in the clause

```
P(f(u,v))<-Q(u):...
```

we delay the transfer of the partial result given by f(u,v) until u has been bound by the execution of Q(u).

Linking the control with the use

In IC-PROLOG, the programmer can restrict the use of the control specified by the text order and annotations of a procedure body to a particular input/output use. He does this by attaching annotations to the terms in the procedure head. If he wants to specify a different control for different uses, he gives a list of control alternatives for the procedure written as a list of clauses

```
[C1,C2,...,Ck].
```

Each of the Ci, ignoring the annotations and the order of the procedure calls, is an exact copy of a single clause

```
B<-A1&...An.
```

Declaratively, the list of control alternatives is equivalent to this single implication. Procedurally, it represents just a single alternative for the execution of a procedure call <-B'. If B' unifies with B atmost one of the control alternatives in the list can be used for the execution of the call. Which one is used, is determined by the "^", "?" annotations attached to terms in the procedure head of each copy of the clause.

These annotations specify extra constraints that must be satisfied by the unification of B' and B before that copy of the clause can be used. Roughly speaking, the annotated term "t?" is used to specify that t has the role of an input template, and the annotation "t^", to specify that t has the role of an output template.

More formally:

(1) An occurence of a variable annotated "v?" must have the role of an input argument of the procedure. During the unification this occurrence of the variable must have been matched against, and consequently bound to, a non-variable. In this context, a variable of the call that appears in some partial result generated by a currently suspended

producer is treated as a non-variable[1]. We shall call such a variable a data variable of the call.

(2) A non-variable annotated "t?" must be matched against a term t' that is a substitution instance of t. In other words, t is used to select the components of some incoming data structure.

(3) An occurence of a variable annotated "v^" must be an output argument of the procedure. During the unification this occurence of the variable must have been matched against a variable of the call which is not a data variable.

(4) A non-variable annotated "t^" must been matched against and become the binding of a variable of the call. t is the first approximation to some output binding that will be produced by the execution of this procedure.

Examples

The annotated procedure head in

```
R(x?,z^)<-Q(x,y)&P(y,z)
```

restricts the use of the control regime that executes Q then P to the case when x is given and z is to be found.

The annotations in the head of

```
append((u.x)?,y?,(u.z)^)<-append(x,y,z)
```

restrict the use of the procedure to calls where the pattern u.x is used to select the head and tail of the list given as the first argument, where y is bound to a non-variable, and where u.z gives the first approximation to the final output of the execution of the procedure.

Selection rule

When an annotated procedure is one of a list of control alternatives it can be used only when the call unifies with its head atom and all the extra constraints specified by its head annotations are satisfied. The first alternative that can be used is the one which is used. If the unification succeeds, but none of the input/output constraints of the control alternatives is satisfied, there is a control error.

2.3 Changing the control

We shall exemplify the sort of computational changes we can bring about by varying the control component of a logic algorithm by looking at different control specifications for the example programs of Chapter 1. We shall use the control notation of IC-PROLOG, and assume back-

---

[1] The variable will actually be bound to a process descriptor recording a currently active consumer.

tracking search.

Back-tracking as iteration

The data base program of Chapter 1 comprised a set of assertions about the fathered and married relations. The order in which these assertions are written defines their try order, in other words, it gives the order in which the assertions will be searched when they are accessed by the calls of the goal clause

<-Tom fathered x & x married Jane.

This goal clause, coupled with the computation rule control implicit in the order of its calls, defines the search algorithm

```
for each <u,x> listed as u fathered x
  if u=Tom,
     for each <x',v> listed as x' married v
        if x'=x & v=Jane, exit with x;
fail.
```

The left to right order of the calls determines the nesting of the iterations. If we change the order, we invert the loops. Thus, the logically equivalent goal clause

<-x married Jane & Tom fathered x,

defines the retrieval algorithm

```
for each <x',v> listed as x' married v
  if v=Jane,
     for each pair <u,x> listed as u fathered x
        if u=Tom & x=x', exit with x;
fail.
```

The second algorithm will generally find the binding for x more quickly. Since Jane can have at most one husband the inner loop will be executed at most once. In the first algorithm, unless the son of Tom married to Jane is listed first, its inner loop will be executed more than once. Execution time is one of the factors the logic programmer must take into account when he considers what control to specify for his logic program.

Effect of indexing

Both the above algorithms can be significantly modified by indexing the assertions. Moreover, the extent of the indexing will effect which ordering of the calls of the goal clause defines the better retrieval algorithm. If only the fathered relation is indexed, say on the first argument, we should use

<-Tom fathered x & x married Jane.

With the search control given by this level of indexing, this ordering

of the calls gives us the retrieval algorithm

```
for each x such that Tom fathered x
  for each <x',v> listed as x' married v
    if v=Jane & x=x', exit with x;
fail.
```

The restriction of the first loop to those x's that Tom fathered reflects the fact that the evaluator now has direct access to this sublist of the fathered assertions. Of course, we have paid for this faster execution time by the space overhead of the index. Decisions about such trade-offs are part of the pragmatics of the logic + control approach to algorithm construction. But note that the switch to use the faster algorithm is painless to implement. The programmer just requests indexing on the first argument of the fathered relation.

If the married relation is also indexed, say on both arguments, the evaluation of the above goal clause corresponds to the retrieval algorithm

```
for each x such that Tom fathered x
   if x married Jane, exit with x;
fail.
```

The test "x married Jane" has replaced the search over the list of married assertions because checking if the given x is married to Jane is an index access that involves no search.

Giving control alternatives

Suppose that we extend the data base program by adding the clause

x is grandfather of z <- x fathered y & y fathered z.

The retrieval and test algorithm represented by this PROLOG procedure will give acceptable behaviour for the cases where x and z are both given, or when just x is given. However, when z is given and x is to be found, it is not very efficient. It will iterate through all the fathered assertions, and for each child check if it is the father of the given z. To do this, it must again iterate through the entire list of assertions. The retrieval time is of the order n^2. It would be better to find the father of z first, doing one search through the list of assertions, and then find the father of the father of z by one more search. The retrieval time is then of the order 2n.

What we need is an order of execution dependent upon the input/output use. We specify this by the list of control alternatives

```
[x? is grandfather of z <- x fathered y & y fathered z,
 x^ is grandfather of z <- y fathered z & x fathered y ].
```

The control for the case when both x and z are to be found is covered by the second control alternative. For this case, either ordering of the calls gives us much the same retrieval algorithm.

Using most general answers

The logic program for the "front" predicate comprised the clauses:

```
front(n,x,z)<-length(x,n)&append(x,y,z)

length(Nil,0)<-
length(u.x,s(n))<-length(x,n)

append(Nil,x,x)<-
append(u.x,y,u.z)<-append(x,y,z)
```

With the control implicit in the text order of these clauses, its execution for calls in which both n and z are given corresponds to the algorithm

```
construct a list of n variables and assign to x;
until x=Nil,
  if z=Nil, fail
  else assign the head of z to the head of x;
repeat with <tail of x,tail of z> as <x,z>
```

This was the computational use we described when we introduced the program. The execution of the call length(x,n) constructs the list of n variables. The computation of the logic algorithm will be essentially iterative because of the tail recursive calls.

Back-tracking with recursion

By changing the execution order of the calls of the "length" procedure we can effect a radical change of algorithm. By using the logically equivalent clause

```
front(n,x,z)<-append(x,y,z)&length(x,n),
```

we implicitly define the following algorithm for finding the front n elements of a list z.

```
assign <Nil,z> to <x,y>;
until y=Nil
  if length(x)=n, exit with x
  else repeat with <x,y> changed from <...Nil,a.l> to <...a.Nil,l>;
fail.
```

The initialisation of x,y to Nil,z corresponds to the attempt to use the "append" assertion to decompose z. It is always used first because of the try order of the clauses. For each candidate decomposition the execution of the program will check the length of x. If it is n, the current binding for x is the answer. If not, the failure of the "length" call leads to a back-track on the last execution step of the "append" call. This last step will have used the "append" assertion. The undoing of the step will change the currently represented value for x to a term of the form ....w by undoing the Nil binding of w. It will also undo the binding y/a.l for y. The subsequent use of the recursive "append" clause will bind w to a.w', and the next step, which will use the "append" assertion, will bind w' to Nil and y to l. The net effect

of the back-tracking is to change the values of x and y from a pair of bindings of the form <....Nil,a.l> to the pair <...a.Nil,l>.

Using data flow coroutining

The major shortcoming of the above algorithm is that the length of x is recomputed for each new binding. What we should be doing is counting down on the length argument each time we make use of the recursive clause for "append". Exactly this behavior will result if we coroutine the contruction of the candidate bindings for x, given by the call append(x,y,z), with the execution of length(x,n). We specify this by using the annotated clause

```
front(n,x,z)<-append(x,y,z)&length(x?,n).
```

The execution of the "append" and "length" calls are now interleaved. Following each use of the recursive clause for "append" there is a use of the recursive clause for "length". This counts down on the length by generating a recursive call in which the length argument is reduced by 1. The attempt to execute this recursive call generates a data request for a further instantiation of x. The resumed execution of the "append" call will first try to use append(Nil,y,y)<-. This will bind the variable of the last partial result to Nil. The resumed length execution can only continue by using length(Nil,0)<-. The attempt to use this assertion is a check to see if the length argument of the suspended call is 0. If it is not, the attempt to use the clause fails. Back-tracking on the last "append" evaluation step leads to the use of its recursive procedure. This triggers another data transfer to the suspended "length" execution, which again counts down on the length by 1. In essential respects, we have the behaviour of

```
assign <Nil,z> to <x,y>;
until y=Nil
  if n=0, exit with x
  else repeat with <x,y,n> changed from <...Nil,a.l,p> to
<...a.Nil,l,p-1>;
fail.
```

2.4 A case study in algorithm construction

The above examples illustrate the fact that significant changes of behaviour can result from minor changes to the syntax of a PROLOG program. Moreover, these syntax changes only effect the control component of the logic + control combination implicit in the text of the program. They in no way change the logic component, which is the declarative reading of the clauses. This enables us to adopt a two pass approach to the development of a logic algorithm as expressed in a PROLOG program. In the first pass we concentrate on the logic component. Concerning ourselves primarily with the declarative reading, we piece together a Horn clause description of the relations we want to compute. We then address the issue of its computational use. We see if we can re-order and annotate the text of the clauses so that, as a PROLOG program, it express a reasonable logic algorithm.

We shall exemplify the two pass approach to the construction of a logic algorithm by developing an IC-PROLOG program for the eight queens problem.

Describing a solution to eight queens problem

The answer to the problem will be some term that is computed as the answer binding for a variable. The term must denote a configuration of eight queens on a chess board. So the first task is to find some way of naming such configurations.

We can simplify our task if we only consider configurations in which each row and column contains only one queen, a constraint that we know must be satisfied by any solution to the problem. A list of the numbers 1 to 8, in some arbitrary order, can be used to name such a configuration of queens. We take the ith number on the list as the column number of the queen in the ith row.

Under this naming convention, the set of permutations of the numbers 1 - 8 names the set of all the queen configurations that are candidate solutions of the problem. A solution is one of these permutations that names a safe configuration, a configuration in which no queen is on a diagonal with any other. Thus, a safe configuration of queens is an instance of the relation named by Queen-sol(1.2....8.Nil,y), where Queen-sol is the relation described by the clause

Queen-sol(x,y)<-Perm(x,y) & Safe(y).

"Perm" names the list permutation relation, and "Safe" names the unary relation which is true of a list of numbers when it names a safe placing of queens on consecutive rows.

We must give Horn clause descriptions of the Perm and Safe relations. The empty list is a permutation of itself. A list u.x is a permutation of a list v.z if v is on the list u.x, and z is a permutation of u.x

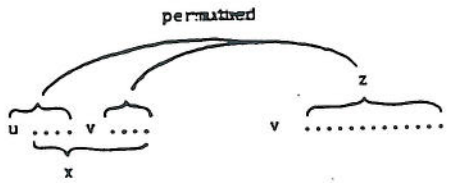


Fig 2.1

with v removed. Fig. 2.1 illustrates this recursive definition. It is embodied in the two Horn clauses:

Perm(Nil,Nil)<-

Perm(u.x,v.z)<-delete(w,u.x,y) & Perm(y,z).

"delete" names the relation that is true when y is the list u.x with

element v removed. The clauses:

delete(v,v.x,x)<-

delete(v,u.x,v.y)<-delete(v,x,y)

are a Horn clause description of this relation.

We now turn to the description of the Safe condition. An empty list of numbers represents a safe placing of queens on consecutive rows. A constructed list u.x names a safe placing if x names a safe placing, and, taking into account that u is the column position for a queen one row away, a queen in column u cannot take any of the queens placed in accordance with the list x. This gives us the assertion and the implication:

Safe(Nil)<-

Safe(u.x)<-no-take(u,x,1) & Safe(x).

Finally, we need to describe the no-take relation. This is the relation satisfied by a column position, u, a list of column positions for consecutive rows, x, and a number, n, when a queen placed in column u, n rows away, cannot take any of the queens placed according to the list x. It is described by the set of clauses:

no-take(u,Nil,n)<-

no-take(u,v.x,n)<-no-diagonal(u,v,n) & no-take(u,x,s(n))

no-diagonal(u,v,n)<-v>u & v=u+w & w≠n

no-diagonal(u,v,n)<-u>v & u=v+w & w≠n

Here, ">", "..=..+.." and "≠" are assumed as primitive arithmetic predicates. The "no-diagonal" predicate names the relation that includes u v and n when a queen placed in column u, n rows away, is not on a diagonal with one placed in column v.

The above clauses constitute a Horn clause axiomatisation of the concept of a solution to the eight queens problem. Let us now turn our attention to their computational use.

Transformation into a logic algorithm

With an eye on this computational use, we have already chosen a suitable ordering of the calls of each clause. We wrote the Queen-sol clause as

Queen-sol(x,y)<-Perm(x,y) & Safe(y)

so that the Perm call would be used to generate the candidate queen configurations which are then tested by the Safe call. However, there is a problem with this generate and test method. Each time a permutation is generated that is not safe, the back-tracking to produce a modified permutation will lead to the loss of all the Safe computation

for the previous permutation. To save some of this, we must let the Safe(y) call have the permutation a bit at a time. We need to annotate the clause

```
Queen-sol(x,y)<-Perm(x,y) & Safe(y?).
```

Let us see what computation will result. The procedures that will be invoked by these two calls are:

```
Perm(u.x,v.z)<-delete(v,u.x,y) & Perm(y,z)

Safe(u.x)<-no-take(u,x,1) & Safe(x).
```

A data transfer to the Safe(y?) call will take place immediately after the unification of the call Perm(x,y) with Perm(u.x,v.z) because this will bind y to the template "v.z". It is much better if we delay the data transfer until the value of v has been computed by the delete(v,u.x,y) call. We do this by changing the "&" of the Perm procedure to a ":".

Now let us consider what will happen when the Safe(x?) call is executed as a result of this first data transfer. The sequential control implicit in the text order of the procedure means that the no-take(u,x,1) call will be executed first. At this point u will be bound to the column position for the first queen, but x is the tail list of queen positions that remain to be computed. The attempt to execute no-take(u,x,1), using either of the "no-take" procedures, will result in a data request for the first element of this tail list of queens.

When this data request has been satisfied, execution of the no-take(u,x,1) resumes. Its execution will generate a sequence of data requests for each of the elements on the list x. It must have all the queen positions on the list in order to check that the first queen is not on a diagonal with any of them. During this interaction there will ofcourse be back-tracking. Each time a new queen can be taken by the first queen, the last slice of the Perm computation is undone. However, there is no check that the newly placed queen cannot be taken by some other queen already positioned. This no-diagonal check is embedded inside the recursive call to the Safe procedure. It will only be executed after the execution of all the no-diagonal checks for the first queen have been completed.

The ideal behaviour would be to execute the two calls of the

```
Safe(u.x)<-no-take(u,x,1)&Safe(x)
```

procedure in parallel, or rather pseudo-parallel, since we have assumed a one call at a time evaluator. When we start the execution of the no-take check for queen position u, we should also start to check that the tail list x is safe. Only when both of these sub-computations are suspended due to data requests, would we suspend the parent call. With this control regime, each time a new queen position is computed, the resumed execution of the top level Safe call will result in the parallel execution of a sequence of no-take checks. Each of these is the no-take check for one of the queens already placed. Each was suspended waiting for the new queen placing. At the same time, a new no-take execution is

started. This will check that the new queen cannot take any of the queens that have yet to be placed.

Such a parallel control regime is feasible to implement. As a possible annotation, we could replace the "&" separator of a sequence of calls that should be executed in parallel by a "//". So, the Safe procedure would be re-written

```
Safe(u.x)<-no-take(u,x,1)//Safe(x).
```

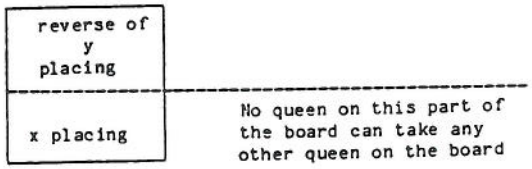
Unfortunately, such a program could not be executed as an IC-PROLOG program. It makes use of a control facility that is outside the control set that we can use.

No matter, we can use this shortcoming of IC-PROLOG to illustrate the fact that we can nearly always compensate for the lack of some control facility by modifying the logic of the program. In this case, we must revise the "Safe" clauses so that a strictly sequential execution will check each new queen against each of the queens already placed.

This behavioural specification leads to the set of clauses:

```
Safe(x)<-Safe-pair(Nil,x)

Safe-pair(y,u.x)<-no-take(u,y,1) & Safe-pair(u.y,x)
Safe-pair(y,Nil)<-
```

At each use of the recursive procedure for "Safe-pair" the first argument of the call is the reverse of the list of queens that have been checked for diagonal takes, and the second argument is the tail list of queens that have yet to be checked. Thus the original list of queens is the reverse of y followed by x. For a declarative reading of the program, we take "Safe-pair" as the name of a binary relation that is true of a pair of lists y and x when reverse(y) concatenated with x names a queen configuration in which no queen on the x part of the board can take any queen on the board (see Fig. 2.2). We leave the reader to



y and x are a Safe-pair of queen placings

Fig.2.2

check that the new "Safe" clause embodies a correct definition of a safe relation in terms of the safe-pair relation, and that the two clauses for "Safe-pair" are both true statements about this new relation. Our modified program, though less obviously correct, still comprises a set

of true statements about the eight queens problem.

The final, revised and annotated program is:

```
Queen-sol(x,y)<-Perm(x,y) & Safe(y?)

Perm(Nil,Nil)<-
Perm(u.x,v.z)<-delete(v,u.x,y) : Perm(y,z)

delete(u,u.x,x)<-
delete(v,u.x,u.y)<-delete(v,x,y)

Safe(x)<-Safe-pair(Nil,x)

Safe-pair(y,Nil)<-
Safe-pair(y,u.x)<-no-take(u,y,1) & Safe-pair(u.y,x)

no-take(u,Nil,n)<-
no-take(u,v.x,n)<-no-diagonal(u,v,n) & no-take(u,x,s(n))

no-diagonal(u,v,n)<-v>u & v=u+w & w≠n
no-diagonal(u,v,n)<-u>v & u=v+w & w≠n
```

The execution of this IC-PROLOG program corresponds to the simple minded back-tracking algorithm which places the queens one at a time on successive rows. It always tries to place the next queen on the left-most free column (because of the try order for the "delete" clauses). If it cannot place a queen on the next row in a position in which it cannot be taken by an earlier queen, it back-tracks and tries to move the previous queen to the right.

Logic vs. control

The above example in algorithm development has illustrated what seems to be a general phenomenon. It is the trade-off between logic and control. To achieve a particular algorithmic behaviour we often have the choice of using a simple, specification-style logic, with an elaborate control, or a complex, more computational logic, with a simple control.

The major advantage of elaborate control facilities is this ability to execute intuitively correct descriptions as algorithms. Kowalski[1979a] elaborates on this point. In his paper entitled "Algorithm=logic+control" he examines the use of a much more elaborate control component than we have considered here. The abstract interpreter he treats is able to mix the top-down or recursive use of the clauses that we have, with a bottom-up use, a use that adds new assertions to the program when the antecedent of some clause has been proved. By using a control notation that can specify the appropriate mix of bottom-up and top-down execution, he shows how very sophisticated algorithmic behaviour can be obtained using a very simple logic.

Chapter 3 Non-procedural Semantics

When we write down the clauses of a logic program we usually have in mind some intended interpretation. This is an interpretation of its constants, functors and predicates as the names of particular objects or individuals, particular functions, and particular relations. Relative to this interpretation, we make sure that each clause is a true statement, that is, we make sure that the intended interpretation is a model of the program. With this same interpretation in mind, we then compose a goal clause so that its conjunction of atoms denotes the relation an instance of which we want to compute.

Each successful evaluation of the goal clause will return an answer substitution

{x1/t1,..,xk/tk}

for all the variables of the goal clause. Relative to our intended interpretation of the program, the tuple of binding terms

<t1,..,tk>

will denote a tuple of individuals, or, if some of the terms contain variables, a set of tuples of individuals. No matter what intended interpretation we had in mind, we would like to be sure that the tuples of individuals named by the answer substitution belong to the relation named by the goal clause.

We can use this naming requirement to give a non-procedural semantics for answer substitutions. We shall say that an answer substitution is correct if it names a set of instances of the relation named by the goal clause for every model of the program.

This is the model theory semantics of answer substitutions that we develop more fully in this chapter. Following van Emdem and Kowalski[1976], we show how it can be recast in the lattice theory framework of the Scott fixpoint semantics. Finally, we investigate its relationship with the procedural semantics of Chapter 1.

3.1 Interpretations and models

In the syntax of logic progams, the logical symbols and the implicitly quantified variables have a fixed meaning. The constants, functors and predicates are 'free' symbols to be used by the programmer as the names of any individuals, functions and relations he has in mind. An interpretation is the mapping from these free symbols to their

denotations.

DEFINITION
An **interpretation** comprises:
(1) A non-empty set D called the domain of the interpretation.
(2) For each constant the assignment of some individual in D.
(3) For each n-ary functor the assignment of some total function from D^n to D.
(4) For each n-ary predicate the assignment of an n-ary relation over D,
if n>0, this is some subset of D^n,
if n=0, it is a 0-ary relation which is truth value, true or false.

There is no requirement that different names be assigned different denotations.

Given an interpretation, each program clause, each goal clause, and each substitution also has a denotation as follows:

Denotation of a program clause

A clause

R(t1,..,tn)<-A1&..&Am

denotes one of the truth values, i.e. it is either true or false. It is true if and only if for every assignment of individuals of D to the variables of the clause, when the antecedent A1&..&Am is true, so is the consequent R(t1,..,tn).

A conjuction A1&..&Am is true if and only if each atom of the conjunction is true.

An atom R(t1,..,tn) is true if and only if
(i) for n=0, the predicate R has the value true,
(ii) for n>0, the tuple of individuals denoted by <t1,..,tn> is in the extension given to R.

The individual denoted by a term which is a variable is the individual assigned to the variable, that denoted by a term which is a constant is the individual assigned to the constant, and that denoted by a term f(t1,..,tk) is the value of the function assigned to f for the tuple of arguments denoted by <t1,..,tk>.

Denotation of a goal clause

The denotation of a goal clause

<-B1&..&Bn

is the denotation of its conjunction of atoms. This is a k-ary relation, where k is the number of variables in the clause.

Let x1,..,xk be these variables in lexicographical order. The relation denoted by the clause is:

(i) for k>0, the set of tuples

{<e1,..,ek> : for the assignment xi=ei, 1≤i≤k, B1&..&Bn is true}

(ii) for k=0, the truth value denoted by the conjunction of atoms B1&..&Bn

Denotation of a substitution

Let <t1,...,tk> be the tuple of binding terms of a substitution ordered by the lexicographical ordering of the variables x1,...,xk for which they are the bindings. The substitution denotes the k-ary relation

{<e1,..,ek> : <t1,..,tk> denotes <e1,..,ek> for some assignment to its variables}.

DEFINITIONS
(1) A **model** of a program P is an interpretation for which each clause is true.
(2) An answer substitution s for a goal clause C and program P is **correct** if for every interpretation that is a model of the program the relation denoted by s is included in the relation denoted by C. For the special case of a goal clause without variables, the answer **true** is **correct** if C is true for every model of P.
(3) Two answer substitutions (for the same variables) are **equivalent** if they denote the same relation in all interpretations.

The definition of a correct answer substitution could be reformulated as:

(A) for every model of P, Cs is true for every assignment to its variables.

or, as:

(B) the universal closure of Cs is a **logical consequence** of P.

The reformulation as (A) relies on the fact that s denotes a sub-relation of C if and only if Cs is true for every assignment to its variables. The re-formulation as (B) appeals to the standard model theory definition of logical consequence. The universal closure of Cs is (∀y1,..,yn)Cs, where y1,...,yn are all the variables of Cs. We shall generally use formulation (B).

The definition of equivalence of answer substitutions is the standard model theory definition of logical equivalence.

Example models

We shall describe three different interpretations, each of which is a

model of the program

```
append(Nil,x,x)<-
append(u.x,y,u.z)<-append(x,y,z).
```

For each model we shall give the denotation of the goal clause

```
<-append(x,y,A.B.Nil),
```

and the denotation of each of the answer substitutions

```
s1={x/Nil,y/A.B.Nil}
s2={x/A.Nil,y/B.Nil}
s3={x/A.B.Nil,y/nil}.
```

These are all the correct answers for the goal clause and program.

Interpretation 1

Domain is the set of natural numbers.

Among the assignments to the constants,functors and predicates:

"Nil","A","B" are 1,2,3 respectively.

"." is the multiplication function.

"append" is the relation {<m,n,p> : mxn=p}.

For this interpretation the assertion of the "append" program is the statement that

for all numbers n, 1xn=n,

and the implication is the statement that

for all m,n,p,i, if mxn=p then (ixm)xn=ixp.

In other words, the assertion is the true statement that 1 is a multiplicative identity, and the implication is the true statement that multiplication is associative. The interpretation is a model of the program.

The goal clause denotes the relation

{<m,n> : mxn=2x3x1} = {<1,6>,<2,3>,<3,2>,<6,1>}.

s1 denotes the singleton set of pairs {<1,6>}.

s2 denotes the singleton set of pairs {<2,3>}.

s3 denotes the singleton set of pairs {<6,1>}.

As required each substitution denotes a sub-relation of the goal clause relation.

Interpretation 2

Domain is a set of objects D that includes some base set of objects B, and is closed under an object constructor function. The constructor, cons, takes a pair of objects o1,o2 into an object o3 which is different from o1 and o2 and unique to each pair of arguments.

One of the base objects is the empty list []. All the other base objects are called atoms. A special subset of the constructed objects is the set of finite length lists {[o1,..,ok] : k>0}. The list [o1,..,ok] is the value of cons(o1,[o2,..,ok]).

Among the assignments to the constants,functors and predicates:

"Nil" is the empty list [].

"A" and "P" and all the other constants are each assigned a unique atom. Since each constant denotes a different atom, we can identify the atom with the constant.

"." is the constructor function cons.

"append" is the relation

{<[],o,o> : o in D}
union
{<[o1,..,oj],[oj+1,..,ok],[o1,....,ok]> : o1,..,ok in D}
union
{<[o1,..,ok],o,cons(o1,cons(o2,..,cons(ok,o)> : o,o1..,ok in D}

This is the intended intepretation for the program that we referred to informally in Chapter 1. We leave the reader to check that it is model of the program.

The relation denoted by the goal clause comprises the three pairs of lists <[],[A,B]>, <[A],[B]>, <[A,B],[]>. These are the denotations of the binding terms of s1,s2 and s3 respectively. For this model, the three answer substitutions exactly cover the goal clause relation.

Interpretation 3

Domain is the set of strings that are the terms of our logic program syntax.

Each constant c is assigned the term c.

Each n-ary functor f is assigned the function that takes the tuple of terms <t1,..,tn> into the term f(t1,..,tn). Thus, "." is the function that takes "A" and "Nil" into ".(A,Nil)". (Remember that A.Nil is just syntactic sugar for ".(A,Nil)").

"append" is the relation

{<Nil,t,t> : t a term}
union
{<.(t1,.(t2,—,.(tk,Nil)--),t,.(t1.(t2.—.(tk,t)--)> :
t,t1,..,tk terms}

We leave the reader to check that this interpretation is a model. The goal clause denotes the relation

{<Nil,.(A,.(B,Nil))> , <.(A,Nil),.(B,Nil)>, <.(A,.(B,Nil)),Nil>}.

Each pair in the relation is denoted by the pair of binding terms of exactly one of the answer substitutions.

Covering the goal clause relation

For the last two interpretations the three answer substitutions s1,s2,s3 have together covered the corresponding goal clause relation. To cover the goal clause relation for the first interpretation we should need to add the substitution

s4={x/B.Nil,y/A.Nil},

which denotes the extra tuple <3,2>. But the binding terms of this substitution do not denote an instance of the goal clause relation for the other two models of the program. A set of correct answer substitutions does not necessarily cover the goal clause relation for each model. The extent of the cover is constrained by the fact that each substitution must denote a relation that falls inside the goal clause relation for every other model.

3.2 Herbrand interpretations

The last interpretation given above is an example of a special class of interpretations called Herbrand interpretations.

DEFINITION
A **Herbrand interpretation** is an interpretation with domain the set of terms which gives the constants and functors their free interpretation. That is,
(i) each constant c is assigned itself,
(ii) each k-ary functor f is assigned the function that takes any k-tuple of terms <t1,..,tk> into the term f(t1,..,tk).

Herbrand interpretations differ only with respect to the extensions assigned to the predicates. Moreover, the extension assigned to a predicate R can be specified by the set of atoms

{R(t1,..,tn) : <t1,..,tn> is in the relation assigned to R}.

We can equate the set of all Herbrand interpretations with a set of all sets of atoms.

Herbrand representatives

The set of Herbrand interpretations is of special importance because they are the only interpretations that we need consider when determining the correctness of an answer substitution. In this matter, they represent the set of all interpretations. First, we define a mapping H which takes an arbitrary interpretation into its Herbrand representative.

DEFINITION
The function H from interpretations to Herbrand interpretations is such that

H(I) = {R(t1,..,tn) : the universal closure of R(t1,..,tn) is true for interpretation I}.

Theorem 3.1

If I is a model of some program P, then H(I) is a model of P.

Proof

Suppose that I is a model of P, but H(I) is not. We shall derive a contradiction.

If H(I) is not a model of P then there is some clause

B<-A1&..&Am

in P which is false for H(I). That is, for some assignment x1=t1,...,xk=tk of terms to the variables of the clause, the conjunction A1&..&Am is true but B is false.

The assignment of terms to the variables of the clause is a substitution s. B will be false for the assignment only if Bs is not in the atom set of H(I), and the antecedent conjunction will be true for the assignment only if each of the atoms of [A1&..&Am]s is in the atom set.

By definition of H(I), if Bs is not in the atom set then there is some assignment y1=e1,...,yi=ei of elements from the domain of I to the variables of Bs such that Bs is false for this assignment. If we extend this to an assignment y1=e1,...,yn=en to all the variables in [B<-A1&..Am]s, we have an assignment for which Bs is still false, but for which each of [A1]s,...,[Am]s is true. Each [Ai]s is true for this assignment because it is in the atom set of H(I). It must, therefore, be true for every assignment to its variables.

Now,

B{x1/t1,..,xk/tk} is false and [A1&..&Am]{x1/t1,..,xk/tk} is true

for the assigment yi=e, 1≤i≤n, only if,

B is false and A1&..&Am is true

for the assignment xi=denotation of ti, 1≤i≤k.

In which case, the clause B<-A1&..&Am is false for interpretation I. This contradicts the assumption that I is a model of the program.

Theorem 3.2

The universal closure of Cs is true for an interpretation I if and only if it is true for interpretation H(I).

Proof

<= Assume that, for H(I), Cs is true for every assignment of terms to its variables.

Suppose y1,..,yn are the variables of Cs. Since Cs is true for every assignment of terms to these variables, it must be true for the assignment y1=y1,...,yn=yn. In other words, Cs must be true of the interpretation H(I). This means that each of the atoms of Cs is in the atom set of H(I). By definition of H, this is the case only if the universal closure of Cs is true for interpretation I.

=> We leave the proof of this implication for the interested reader. It is just as straightforward.

Equivalent interpretations

The function H induces an equivalence relation on interpretations, two interpretations being in the same equivalence class if they map into the same Herbrand interpretation. Since H maps a Herbrand interpretation into itself, each equivalence class contains exactly one Herbrand interpretation which we can take as the representing element of the class.

The above theorem tells us, that with respect to the correctness requirement for answer substitutions, the interpretations of each equivalence class are equivalent.

This, together with the fact that H maps models into models, gives us the following theorem.

Theorem 3.3

An answer substitution s is correct if and only if the universal closure of Cs is true for every Herbrand model of the program.

Proof

=> If s is correct, the universal closure of Cs is true for all models, hence for all Herbrand models.

<= Let M be any model. Theorem 3.1 tells us that its Herbrand representative H(M) is also a model. By assumption, the universal closure of Cs is true for model H(M). Hence, by Theorem 3.2 it is also true for the arbitrary model M.

3.3 Fixpoint semantics

Following van Emden and Kowalski[1976] we can recast the model theory semantics in the Scott lattice theory framework [Scott 1970]. To do this, we must interpret each logic program P as an equation

$$x = P(x),$$

where P is a continuous function over some complete lattice of the possible denotations of P. The least fixpoint of P, i.e. the least solution of the above equation, is taken as the denotation of P. We give a brief summary of the key concepts of the fixpoint approach.

DEFINITIONS
(1) A **complete lattice** is a set S over which there is a reflexive, antisymmetric, transitive order relation, $\leq$. For each subset X of S there is a least upper bound, lub(X), in S.
(2) A function P over a complete lattice S is **continuous** if for every directed subset X of S

$$P(lub(X)) = lub\{P(x) : x \text{ in } X\}$$

(3) A **directed** set is a set which contains an upper bound for each of its finite subsets.

The fact that a complete lattice contains a lub for every subset implies that there is a top element, T, and a bottom element, $\perp$. It also implies that every subset X has a greatest lower bound, glb(X), in S.

That P is continuous implies that it is monotonic, i.e. that

$$x \leq y \text{ implies } P(x) \leq P(y).$$

Hence, by Tarski's fixpoint theorem for a monotonic function over a complete lattice [Tarski 1955], the least fixpoint of P exists, and is

(A) $glb\{x : P(x) \leq x\}$.

There is a second identification of the least fixpoint. By a generalisation of the Kleene recursion theorem[Kleene 1952], it is

(B) $lub\{P^i(\perp) : i \geq 0\}$.

For a proof, see [Stoy 1977 p.112]. It appeals to the continuity property of P.

Lattice of Herbrand interpretations

Given a logic program P we can use it to compute a set of answer substitutions for every goal clause of the form <-R(x1,..,xk), R a k-ary predicate. Each set of answer bindings defines a Herbrand extension for R, and by computing a Herbrand extension for each predicate we compute a Herbrand interpretation. So, as the set S of candidate denotations for the program we take the set of all Herbrand interpretations.

Under the partial order relation

$$I \leq I' \text{ iff the atom set of I is included in that of I',}$$

they are distributed over a complete lattice S. The least upper bound of any subset Y of S is the atom set which is the union of the Herbrand interpretations in X. The greatest lower bound of X is the intersection of its Herbrand interpretations. The top element of the lattice is the Herbrand interpretation comprising the set of all atoms, the bottom element is the Herbrand interpretation with the empty set of atoms.

We re-interpret the program P as the equation

$$I = P(I)$$

where P is the function

$$P: I \to \{B' : B' \leftarrow A'1 \& \ldots \& A'm \text{ is an instance of a clause in P such that } A'1,\ldots,A'm \text{ are all in I}\}.$$

We shall call P the immediate consequence function for the program. If I defines the extension of each relation accessed by the procedure calls of a program clause, then P(I) is the interpretation that defines the extension of each relation that can be computed by the program. This accords with the conventions of the fixpoint approach, and is consistent with the re-interpretation of the program as the equation I=P(I).

Theorem 3.4

P is continuous.

Proof

Let X be a directed set of Herbrand interpretations. To prove that P is continuous we must show that an atom B' is in P(lub(X)) if and only if it is in $lub\{P(I): I \text{ in } X\}$.

B' is in P(lub(X))

iff B'<-A'1&..&A'm is an instance of a clause in P such that A'1,...,A'm are in lub(X) (definition of P)

iff B'<-A'1&..&A'm is an instance of a clause in P such that A'1,...,A'm are in interpetations I1,...,Im of the set X
(lub(X) is the union of the I in X)

iff B'<-A'1&..&A'm is an instance of a clause in P such that A'1,...,A'm are in some I in X
(the if-half is trivial, for the only-if half I is the lub of I1,...,Im. Since X is directed, and I is the lub of a finite subset of X, I must be in X.)

iff B' is in P(I) for some I in X (definition of P)

iff B' is in lub{P(I): I in X} (definition of lub).

The lattice S contains every Herbrand interpretation. For the model theory semantics only those interpretations which are models are of interest. The following theorem characterises the Herbrand models in terms of the function P.

Theorem 3.5

A Herbrand interpretation I is a model of a program P iff $P(I) \leq I$.

Proof

A Herbrand interpretation I is a model of P

iff for each clause B<-A1&..&Am in P, and for every substitution s for its variables, whenever [A1&..&Am]s is true Bs is true

iff for every substitution s, whenever each of [A1]s,..,[Am]s are in I, Bs is in I

iff I contains all the atoms in P(I)

iff $I \geq P(I)$.

Denotation of a program

DEFINITION
The least fixpoint of P is the **fixpoint denotation** of the program P.

Theorem 3.6

The fixpoint denotation of a program P is the Herbrand model which is the intersection of all the Herbrand models of P.

Proof

The fixpoint denotation of the program is an interpretation I such that $I \geq P(I)$. By Theorem 3.5, it is a model of the program.

To prove that it is the intersection of all the Herbrand models we use the Tarski identification of the least fixpoint. It is

glb({I: $I \geq P(I)$}) = intersection of {I: I is a Herbrand model of P}.

Expressed more intuitively, the fixpoint denotation of the program is the Herbrand interpretation that assigns the least extension to each predicate compatible with the interpretation's being a model of the program.

Correct answers

The fixpoint denotation of a program P is a single Herbrand model. An answer substitution s must be deemed correct if the universal closure of Cs is true for this particular model. This gives us an alternative

definition of correctness.

DEFINITIONS
An answer substitution s is **fixpoint correct** for a goal clause C and program P if the universal closure of Cs is true of the fixpoint denotation of P.

The following theorem tells us that this is exactly the model theoretic characterisation of a correct answer recast in the fixed point framework.

Theorem 3.7

An answer substitution s is fixpoint correct iff it is correct

Proof

s is fixpoint correct

iff the universal closure of Cs is true for the fixpoint denotation of P

iff it is true for the Herbrand model which is the intersection of all the Herbrand models of P (Theorem 3.6)

iff it is true for every Herbrand model of P

iff s is correct (Theorem 3.3).

Because of this equivalence of the model theoretic and fixpoint definition of correctness we can use either as our non-procedural semantics for answer substitutions. From now on we shall generally appeal to the fixpoint semantics. Note that we have yet to make use of the identification

$$\mathrm{lub}\{P^i(\bot): i \geq 0\}$$

of the least fixpoint of P. We shall use this in proving the equivalence of the procedural and non-procedural semantics, the usual use for the Kleene identification of the least fixpoint.

3.4 Relation to the procedural semantics

The key intermediary in our proof of the equivalence of the procedural and non-procedural semantics is the concept of a complete proof tree as defined in Chapter 1. Remember that a complete proof tree is the object constructed by a successful evaluation.

First, we shall prove that a correct answer substitution is the answer substitution 'displayed' by the goal clause atoms of a complete proof tree. We shall prove that a substitution s is correct for C and P if and only if the atoms of Cs are the goal clauses atoms of a finite complete proof tree for C and P. From this it follows immediately that every computed answer is correct. However, it does not follow that every correct answer can be computed, for the finite proof tree that

displays s may not be a tree that can be constructed by some evaluation.

To prove that every correct answer answer can be computed, we show that the set of finite proof trees are the set of substitution instances of the R-constructed proof trees, R any computation rule. It follows from this that every correct answer is a substitution instance of an R-computable answer.

Theorem 3.8

An answer substitution s is correct if and only if the atoms of Cs are the goal clause atoms of a finite complete proof tree for C and P.

Proof

We reduce the proof of the theorem to a definition and three lemmas.

DEFINITION
An atom B' is an n-level consequence of a program P if B' is in $P^n(\bot)$ for some $n \geq 0$.

Remember that P is the immediate consequence function associated with the program, and $\bot$ is the empty set of atoms.

The 1-level consequences are all those atoms that are substitution instances of the assertions in P.

The n+1 level consequences are the n-level consequences together with the set of atoms

{B': B'<-A'1&..&A'm is an instance of a clause in P
and A'1,..,A'm are n-level consequences}.

The following lemma can be established by a simple induction on n.

Lemma 1

An atom is an n-level consequence iff all its substitution instances are n-level consequences.

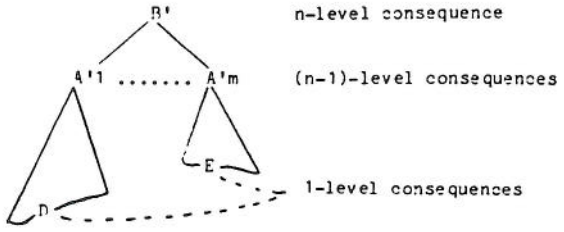
Proof tree display

We can display the derivation of an atom B' in $P^n(\bot)$ as a tree of height at most n. Such a tree is depicted in Fig. 3.1.

By making B' the single descendent of an unlabelled root node the tree becomes a complete proof tree for the goal clause <-B' and the program P. Since it is of finite height, by König's lemma it has a finite number of nodes. Hence:

Lemma 2

An atom B' is an n-level consequence of P iff there is a finite complete proof tree for <-B' and P.



Derivation of n-level consequence B'

Fig. 3.1

We know that a substitution s is correct iff the universal closure of Cs is true for the fixpoint denotation of P, iff every substitution instance of each of its atoms is in the atom set of the least fixpoint of P.

The following lemma tells us that each instance of an atom B is in the least fixpoint of P if and only if B is an n-level consequence for some n. With lemma 2 it implies the theorem. This is because a complete proof tree for Cs comprises separate complete proof trees for each of its atoms joined to a common root.

Lemma 3

Each instance of an atom B is in the least fixpoint of P iff it is an n-level consequence of P for some n.

Proof

An instance of B is in the least fixpoint of P

iff it is in $\text{lub}\{P^n(\bot)\}$ (identity (B) for the least fixpoint)

iff it is in $P^n(\bot)$ for some n (definition of lub)

iff B is an n-level consequence of P (lemma 1).

Theorem 3.8 has the following immediate corollary:

Corollary to Theorem 3.8

Every R-computable answer substitution is correct

Proof

A successful evaluation using computation rule R constructs a complete proof tree. The answer of the evaluation is the substitution that takes the goal clause C into the goal clause atoms of this tree. By the theorem, the substitution is correct.

In resolution terms, the above corollary tells us that LUSH resolution is sound.

We cannot prove exactly the converse of the above corollary despite the fact that the theorem tells us that every correct substitution is 'displayed' on some finite complete proof tree. This is because not every finite proof tree can be constructed. However, what is true, is that every finite complete proof tree is a substitution instance of a constructed tree.

Theorem 3.9

If T' is a finite complete proof tree then, for any computation rule R, there is an R-constructed proof tree T such that T'=Ts for some substitution s.

Proof

A finite complete proof tree T' for a goal clause <-B1&..&Bn is a tree as depicted in Fig. 3.2(b). The initial proof tree for this goal clause, Fig. 3.2(a), is a constructed tree. We can prove the theorem by showing that, using any computation rule R, the initial proof tree can be extended into a proof tree T that maps onto T' under some substitution s.

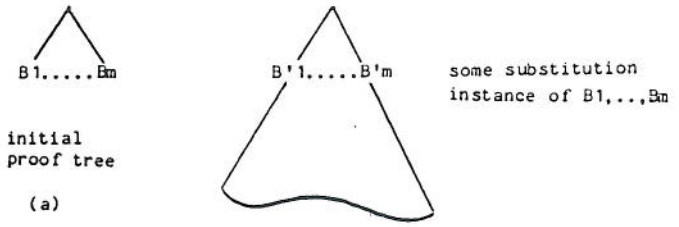


Fig. 3.2

The initial proof tree is just a special case of a constructed proof tree T that maps into T' as depicted in Fig. 3.3. That is, it is a

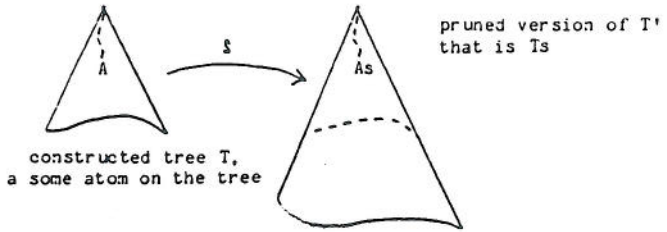


Fig. 3.3

constructed tree such that some substitution s maps it into a pruned version of T'. Let us measure the difference between T and T' by:

number of extra nodes on T' + number of leaf nodes on T' .
not marked as terminal on T

By an induction on the size of this difference, we prove that every constructed tree T that maps into T' can be extended into a complete proof tree that maps onto T'.

Basis

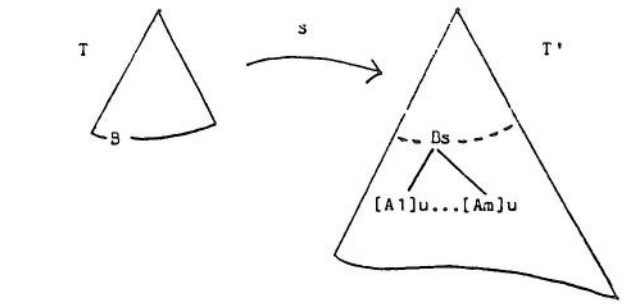
When the difference is zero T is already a constructed complete proof tree that maps onto T'.

Induction step

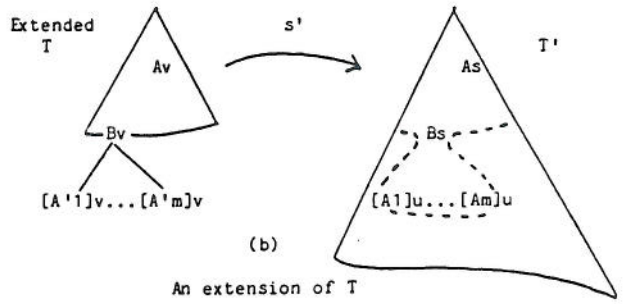
Let the difference between T and T' be k+1. Assume that the extension can be achieved for all T for which the difference between T and T' is k or less.

In extending the tree T the computation rule R will select an leaf atom B of T not marked as terminal. We know that Ts is a pruned version of T'. As depicted in Fig. 3.4(a), there must be an atom Bs on T' that corresponds to the selected atom B.

Since T' is a complete proof tree it must be the case that Bs is the



(a)



(b)

An extension of T

Fig. 3.4

consequent atom of a substitution instance

[B'<-A1&..&Am]u, m≥0

of some clause in the program P. The atoms [A1]u,..,[Am]u will be the immediate descendants of Bs (which is identical to B'u) on T'. Let us call

B'<-A1&..&Am

the validating clause for Bs. A variant

B"<-A'1&..&A'm

of this clause can be used to extend the partially constructed tree T.

Since Bs=B'u, B and the variant B" of B' will be unifiable. Suppose that {x1/y1,..,xk/yk} is the change of variable substitution that takes B' into B". Since B and B" have no variables in common, s union {y1/[x1]u,..,yk/[xk]u} will be a set of bindings for distinct variables.

Applied to B, it will give us Bs. Applied to B", it will give us B'u. Hence, this substitution is a unifier of B and B".

The unification of B and B" will return a most general unifier v. T will then be extended by adding A'1,..,A'm as immediate descendants of B, or, if m=0, by marking B as terminal. The substitution v is then applied to every node on the extended tree (see Fig. 3.4(b)).

The difference between the extended T and T" will be k or less. To apply the induction hypothesis, we must prove that the extended T also maps into T' under some substitution s'.

We know that s union {y1/[x1]u,..,yk/[xk]u} unifies B and B". Since v is a most general unifier of B and B", there is a substitution s' such that

v*s'=s union {y1/[x1]u,..,yk/[xk]u}

s' is the substitution that will take the extended tree into T'.

If s' is applied to each of the new nodes, which are labelled by the antecedent atoms of

[B'<-A1&..Am]{x1/y1,..,x1/y1}*v,

it will give us the antecedent atoms of

[B'<-A1&..&Am]{x1/y1,..,xk/yk}*{y1/[x1]u,..,yk/[xk]u}.

These are the atoms [A1]u,..,[Am]u on T'. Applied to Bv, and any other atom label Av of one of the old nodes, it will produce the atoms Bs and As respectively. Hence, s' maps every atom of the new tree into the corresponding atom on T' as required.

Corollary to Theorem 3.9

For each correct answer substitution s' for a goal clause C there is an R-computable answer substitution s such that Cs'=Cs*s" for some substitution s".

Proof

Since s' is correct, Theorem 3.8 tells us that the atoms of Cs' are the goal clause atoms of some complete proof tree T'. By the above theorem, we know that there is a constructable proof tree T that maps onto T'. The goal clause atoms of T will be the atoms of Cs, where s is the answer computed by the successful evaluation that constructs T. The substitution s" that maps T onto T' will map Cs onto Cs'. Hence, Cs'=Cs*s" as required.

In resolution terms, the above corollary tells us that LUSH resolution is complete. It is a slight generalisation of the normal completeness result which only guarantees the existence of the computable substitution s when the substitution s" has binding terms without variables. Completeness for LUSH resolution was first proved by Hill[1974]. The above proof is somewhat different from the one he gave.

3.5 Independence of the computation rule

Each answer substitution denotes a relation over the Herbrand universe. This is the set of tuples of terms that can be obtained by instantiating the tuple of binding terms of the substitution. Let us call the union of the Herbrand relations denoted by the set of R-computable answers for a goal clauses C, the R-computable extension of C. The following theorem tells us that this is always the extension given to C by the Herbrand interpretation that is the fixpoint denotation of P.

Theorem 3.10

For any computation rule R, the R-computable extension of a goal clause C is the relation assigned to C by the fixpoint denotation of the program.

Proof

We show that each extension includes the other.

The relation assigned to C by the fixpoint denotation of the program is

{<t1,..,tk>: each atom of C{x1/t1,..,xk/tk}
is in the fixpoint interpretation}

= {<t1,..,tk>: {x1/t1,..,xk/tk} is a correct answer}.

That is, a tuple <t1,..,tk> of binding terms is in this relation only if it is the tuple of binding terms of a correct answer. By the completeness corollary of Theorem 3.9, <t1,..,tk> is in the Herbrand extension of an R-computable answer. It follows that the relation assigned to C by the fixpoint interpretation is included in the Herbrand extension of the set of R-computable answers.

By the soundness corollary of Theorem 3.8, each R-computable answer is correct. By the fixpoint definition of correctness, the Herbrand relation denoted by the answer is included in the extension of the relation assigned to C by the fixpoint denotation of the program. So, the Herbrand extension of the set of R-computable answers is included in the extension of this goal clause relation.

Corollary

The computed extension of a goal clause is independent of the computation rule.

The above corollary is our first result concerning the independence of the computation rule. There is a stronger result. We can prove that the different sets of computable answers not only denote the same Herbrand relation, but that they contain essentially the same substitutions.

Let us look again at the proof of Theorem 3.9. In the induction step, where we extended the proof tree T at the selected atom L, we used a variant of the validating clause of the corresponding node on T'. The particular atom that appeared on this node was not relevant. We can replace T' by any other tree T" that has the same validating clause associated with each node. The tree constructed by the inductive proof would be unchanged.

DEFINITION
The proof skeleton **T** of a proof tree T is a structurally identical tree in which each node is labelled by the validating program clause of the corresponding node on T.

We leave the reader to check that in the construction of T implicit in the induction of Theorem 3.9:

(1) The sequence of extension steps that will construct T from the initial proof tree is uniquely determined by the computation rule and the proof skeleton **T'** of T'.

(2) T' could be replaced by any other proof tree with the same skeleton.

(3) The proof skeleton of T is the proof skeleton of T'.

A proof skeleton is what is constructed by the stack implementation described in Chapter 1 if we discard the binding environments. We can summarise the above properties in the following theorem.

Theorem 3.11

For a given computation rule R and proof skeleton **T** there is exactly one successful evaluation for the goal clause C. This constructs a most general proof tree T with proof skeleton **T**. That is, T maps onto any other proof tree with skeleton **T**.

Corollary

An answer substitution s is R-computable only if an equivalent substitution is R'-computable, where R' is any other computation rule.

Proof

The computation of s using rule R will construct a complete proof tree T with the atoms of Cs as the goal clause atoms. Since there is only one successful evaluation corresponding to the rule R and the proof skeleton **T** of T, T must be a most general proof tree for skeleton **T**.

For proof skeleton **T** and rule R' there is also exactly one successful evaluation of C. This will compute an answer s' and in so doing will construct a proof tree T', with the atoms of Cs' as its goal clause atoms. This is also a most general proof tree for skeleton **T**.

Since T and T' are each most general proof trees for skeleton **T**, there is a substitution that maps the atoms of Cs into the atoms of Cs', and vice versa. Cs and Cs' must therefore be variants. If s and s' are the

substitutions

s={x1/t1,..,xk/tk}, s'={x1/t'1,..,xk/t'k},

then, since x1,...,xk are all the variables of C, the term tuples <t1,..,tk>, <t'1,..,t'k> must also be variants. They therefore denote the same relation for every interpretation.

The above corollary is the justification for our claim that the family of algorithms implicit in a given logic program are equivalent. It tells us that each algorithm must compute the same set of answers (modulo equivalence of substitutions).

Chapter 4 Verification

As we stressed in Chapter 1, many logic programs double as their own specification. Reading the clauses of the program as statements about their intended interpretation is often all that is needed to check the truth of each clause, hence to confirm the correctness of the program.

This is not always the case. As we saw with the eight queens program, we are sometimes forced to use less obvious but more computationally useful descriptions in order to obtain a reasonable algorithm. When we do this, we should prove that the program clauses embody a correct description.

There is another problem. Even when the program clauses constitute a correct description, this may be an incomplete description. The set of assertions that are logical consequences of the program may not cover all the instances of some relation that we want to compute. To prove that it does cover all these instances, is to prove that the program description is complete for its intended use.

In this chapter we address the problem of the verification of a logic program. We show how the question of correctness (or completeness) of a program can be reduced to the question of whether some verification sentence, which is a sentence of first order logic, is true of a particular interpretation. Verifying the program is then a matter of proving that the sentence is true of that interpretation. We shall see that the different ways in which we might do this, correspond to different standard methods of verifying programs.

4.1 Verification sentences

Let us try to formalise the notion of correctness of a logic program by looking at an example. Let us consider the "append" program

append(Nil,y,y)<-
append(u.x,y,u.z)<-append(x,y,z),

and its use to append lists. This 'use' is characterised by the set of goal clauses of the form

<-append(t1,t2,z)

where t1 and t2 are both 'list' terms, that is terms of the form t'1.t'2.--.t'n.Nil for n≥0.

The successful evaluation of such a goal clause will construct a complete proof tree for a goal clause atom append(t1,t2,t3). In so doing, it will compute the answer binding z/t3 for z. t3 should name the concatenation of the lists named by t1 and t2. However, suppose that we are only interested in showing that t3 is a list term, that we only want to check that the appending use maps list terms into list

terms. Let us see if we can state this correctness condition more formally.

For the appending use the first two arguments of the call are non-variable terms which belong to the set of pairs of terms

I = {<t1,t2> : t1,t2 are list terms}.

This set is a relation over the Herbrand universe which we shall call the input relation. The third argument of the call is the output variable whose answer binding is sought. Suppose that we use subscripted x's to name the argument positions occupied by the input terms, and subscripted y's to name the argument positions of the output variables. The set of appending calls can be named by the atom/input relation pair

<append(x1,x2,y1),I>.

We shall assume that this denotes the set of goal clauses of the form

<-append(x1,x2,y1){x1/t1,x2/t2,y1/z}

where <t1,t2> is in the relation I and z is a variable not appearing in t1 or t2.

We want to show that the answer binding z/t3 is such that t3 is also a list term. More formally, we want to show that <t1,t2,t3> are in the output relation

O = {<t1,t2,t3> : t3 a list term}.

We know that <t1,t2,t3> is a tuple of terms in the relation

{<t1,t2,t3> : append(t1,t2,t3) is the goal clause atom
of some complete proof tree}.

Let us call this the append relation computed by the program. It follows that the use <append(x1,x2,y1),I> will be correct for the output relation O if it is true that

for all terms t1,t2,t3
<t1,t2> in the relation I
and
<t1,t2,t3> in the append relation computed by the program
implies
<t1,t2,t3> in the relation O.

Using the language of first order logic, this is just the condition that the sentence

(∀x1,x2,y1)[I(x1,x2)&append(x1,x2,y1) -> O(x1,x2,y1)]

should be true of the Herbrand interpretation in which "I" is the input relation, "O" the output relation, and "append" the append relation computed by the program.

Notation

Before we give a general definition of correctness we need some notation.

We shall use **x** to denote a set {x1,..,xm} of m distinct (input) variables and **y** to denote a different set {y1,..,yn} of n distinct (output) variables.

R(**x**,**y**) is the atom R(x1,..,xm,y1,..,yn).

R(**x**^**y**) is some atom obtained from R(**x**,**y**) by permuting its argument variables.

R(**t**^**z**) is R(**x**^**y**){x1/t1,..,xm/tm,y1/z1,..,yn/zn} where z1,..,zk are variables not appearing in **t**=<t1,..,tm>.

I is some m-ary (input) relation over terms.

O is some (m+n)-ary (output) relation over terms.

Correctness

DEFINITIONS
(1) The R-relation **computed** by a program P is the set of term tuples

{<t1,..,tk> : R(t1,..,tk) is a goal clause atom of a
complete proof tree for P}

(2) A **use** of a program is denoted by a pair <R(**x**^**y**),I>. It comprises the set of goal clauses of the form

<-R(**t**^**z**), **t** in the relation I.

(3) The use <R(**x**^**y**),I> is **correct** for an output relation O if the correctness sentence

(∀**x**,**y**)[I(**x**)&R(**x**^**y**) -> O(**x**,**y**)]

is true of the R-relation computed by the program.

In our definition of correctness, we have implicitly assumed that the interpretation for which the correctness sentence is true is the Herbrand interpretation in which the predicates "I" and "O" denote the input and output relations respectively.

Let us check that this definition captures the idea of correctness. The use <R(**x**^**y**),I> is a set of goal clauses of the form

<-R(**t**^**z**), **t**=<t1,..,tm> in I.

We have assumed that the purpose of evaluating such a clause is to compute an answer substitution

{z1/t'1,..,zn/t'n}

that gives us an output tuple t' that corresponds to the input tuple t. To do this, the evaluation of <-R(t^z) must construct a complete proof tree for the atom P(t^t').

The truth of the correctness sentence ensures that

$$I(t)\&R(t\hat{}t') \rightarrow O(t,t')$$

is true of the R-relation computed by the program. But, I(t) is true by assumption, and R(t^t') is true by the 'proof' of the computation of t'. It follows that O(t,t') must be true, i.e. that t' is related to t by the output relation.

Termination

Correctness does not guarantee that some answer tuple for z will be computed. To guarantee this, we must know that some atom R(t^t') is in the R-relation computed by the program whenever I(t) is true. In other words, we must know that

$$\forall x[I(x) \rightarrow \exists y R(x\hat{}y)]$$

is true.

DEFINITION
A program **terminates** for a use <R(x^y,I> if the termination sentence

$$\forall x[I(x) \rightarrow \exists y R(x\hat{}y)]$$

is true of the R-relation computed by the program.

Let us check that the truth of this sentence ensures termination. Consider some goal clause

$$\leftarrow R(t\hat{}z)\ ,\ t \text{ in } I$$

of the use <R(x^y),I>. Since t is in I, the termination sentence tells us that there is some t' such that R(t^t') is the goal clause atom of a complete proof tree. This atom is a substitution instance of R(t^z). By Theorem 3.9 we know that there is a successful evaluation of <-R(t^z) using any computation rule.

Completeness

The truth of both the correctness sentence and the termination sentence guarantees that there is always a successful evaluation of the goal clause

$$\leftarrow R(t\hat{}z),\ t \text{ in } I$$

that computes some t' such that <t,t'> is in the relation O. If we want to make sure that we can compute every t' related to the input t by O,

then the program must be complete for the relation O.

DEFINITION
A use <R(x^y),I> of a program is **complete** for an output relation O if the completeness sentence

$$(\forall x,y)[I(x)\&O(x,y) \rightarrow R(x\hat{}y)]$$

is true of the R-relation computed by the program.

Consider the goal clause

$$\leftarrow R(t\hat{}z),\ t \text{ in } I.$$

We leave the reader to check that the truth of the completeness sentence ensures that every t' related to t by O is an instance of some computable answer binding for z. This is another application of Theorem 3.9.

Total correctness

The correctness sentence can be reformulated as the logically equivalent sentence

(A) $(\forall x,y)[I(x) \rightarrow [R(x\hat{}y) \rightarrow O(x,y)]]$.

Similarly, the completeness sentence can be reformulated as the logically equivalent sentence

(B) $(\forall x,y)[I(x) \rightarrow [O(x,y) \rightarrow R(x\hat{}y)]]$.

The conjunction of (A) and (B) is equivalent to the conditional equivalence

$$(\forall x,y)[I(x) \rightarrow [R(x\hat{}y) \leftrightarrow O(x,y)]].$$

If this sentence is true of the program computed relation, then every term tuple t' in the O relation to some input tuple t can be computed, and, conversely, every computed t' is in the O relation to the input tuple. It is the strongest verification condition.

DEFINITION
A use <R(x^y),I> of a logic program is **totally correct** for an output relation O if the total correctness sentence

$$(\forall x,y)[I(x) \rightarrow [R(x\hat{}y) \leftrightarrow O(x,y)]]$$

is true of the R-relation computed by the program.

The total correctness sentence tells us that under the restriction that certain arguments lie in some relation I, the relation computed by the program is the relation O.

4.2 The theory of the program computed relations

The above definitions have reduced the proof of each of several verification properties to the proof that some sentence is true of a particular interpretation. One of the most straightforward ways of showing that a sentence is true of some interpretation is to show that it can be derived from a set of sentences that are obviously true of that interpretation. As depicted in Fig. 4.1, this means showing that the sentence is a theorem of some theory of the interpretation, the axioms of the theory being the sentences that are unquestionably true.

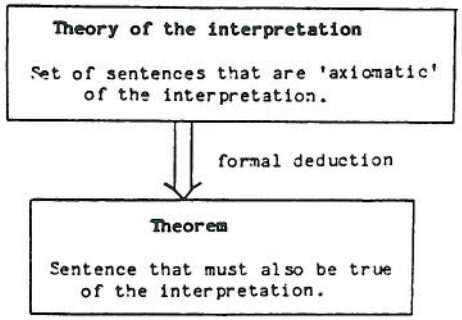


Fig. 4.1

Let us pursue this approach. Let us build up a little theory of the Herbrand interpretation in which a predicate R that appears in a logic program P denotes the R-relation computed by the program. To this theory we add axioms describing the input and output relations I and O. This gives us a theory of the Herbrand interpretation for which a given verification sentence must be true. We can then attempt to prove that the sentence is true by trying to derive it as a theorem of this theory.

We shall assume that the reader is familiar with the idea of a first order theory, and with the language of first order predicate logic with equality. [Mendelson 1964] is a suitable introduction. We also assume some familiarity with natural deduction style inference rules, and the informal presentation of proofs (see [Quine 1964] or [Suppes 1957]).

The identity theory

The interpretation we want to describe is a Herbrand interpetation. This gives the constants and functors their free interpretation. Each

of the following axiom schemas is true of this free interpretation.

```
    c ≠ c',   c and c' different constants
    c ≠ f(x), c any constant, f any functor
 f(x) ≠ g(y), f and g different functors
 t[x] ≠ x,    t[x] any term in which x appears

 f(x) = f(y) -> x = y
```

As with program clauses, each axiom is assumed to be implicitly universally quantified with respect to its variables. We adopt this as a convention for axioms.

The first schema tells us that different constants denote different individuals; the second that the denotation of a constant is never the value of a function; and the third that different functors denote functions with disjoint ranges. The fourth schema tells us that no composition of functions maps an individual into itself. The last tells us that each function is a one-to-one mapping of the domain.

Let us call this set of axiom schemas the identity theory for a Herbrand interpretation. In describing the Herbrand interpretation comprising some set of program computed relations, we can use any axiom of this identity theory.

Equivalence implicit in the program clauses for R

Consider again the "append" program

```
append(Nil,y,y)<-
append(u.x,y,u.z)<-append(x,y,z).
```

The clauses of the program are axioms for the relation it computes. This is because Theorem 3.8 tells us that the append relation computed by the program is the just the relation assigned to "append" in the fixpoint denotation of the program, which is a model of the program.

They are, however, quite weak axioms. They tell us only that this fixpoint interpretation is such that

```
    an atom append(t1,t2,t3) is in the interpretation
if
    t1=Nil and t2=t3
  or
    there are terms t,t'1,t'3 such that t1=t.'1, t3=t.t'3 and
    append(t'1,t2,t'3) is in the interpretation.
```

In terms of the program associated transformation P, the program clauses tell us only that the fixpoint denotation is an interpretation I such that $P(I) \leq I$.

We know that it is an interpretation such that P(I)=I. In other

words, we know that it is an interpretation such that

append(t1,t2,t3) is true

if, and only if,

t1=Nil and t2=t3
or
there are terms t,t'1,t'3 such that t1=t.t'1, t3=t.t'3 and append(t'1,t2,t'3) is true.

Using the language of first order logic, this is the statement that

append(x,y,z) <-> x=Nil&y=z v (∃u,x',z')[x=u.x'&z=u.z'&append(x',y,z')]

is true of the program computed relation.

More generally, suppose that

R(t1,..,tn)<-A1&..&Ap

is some program clause for a predicate R. Let u1,..,um be all the variables of the clause and let x1,..,xn be n variables not appearing in any clause for R. We can reformulate the clause as the equivalent implication

R(x1,..,xn)<-(∃u1,..,um)[x1=t1&..&xn=tn&A1&..&Ap].

Let us call this the general form of the clause.

Now suppose that

R(x1,..,xn)<-E1
.
.
.
R(x1,..,xn)<-Ek

are the k general forms of each of the program clauses about R. To say that the Herbrand interpretation of all the program computed relations is a fixpoint of the program associated transformation P, is to say that

R(t'1,..,t'n) is true

if, and only if,

E1{x1/t'1,..,xm/t'n} is true
or .
.
.
or Ek{x1/t'1,..,xm/t'n} is true.

Hence, the equivalence

R(x1,..,xn) <-> E1 v ... v Ek

is true of the program computed relations to which it refers. We call this the R-equivalence implicit in the set of clauses for R.

Equivalence implicit in a single clause

We can always use the equivalence implicit in the set of clauses for a predicate as an axiom of our theory of the program computed relations. However, when we are interested in proving poperties of a particular use <R(x^y),I>, it is often more convenient to use a set of slightly weaker axioms which are equivalences implicit in the individual clauses.

Each clause about R will be of the form

R(t^t') <- A1&..&Am

where t is some tuple of input templates and t' is some tuple of output templates. Let us suppose that the tuple of input templates is unique to this clause, or, more formally, that the tuple of terms t will not unify with the corresponding tuple of terms of any other clause for R. When this is the case, the equivalence

R(t^y) <-> (∃u1,..,uk)[y=t'&A1&..&Am]

is true of the program computed relations to which it refers. Here, u1,..,uk are all the variables of the program clause that do not appear in t.

We can argue that the equivalence must be true by appealing to the definition of the program associated transformation P and noting that an atom which is a substitution instance of R(t,y) will be in P(I) iff the righthand side of this equivalence is true of I. Not surprisingly, it can also be inferred from the equivalence implicit in the set of clauses for R using the identity theory.

The following example will illustrate this derivation of an equivalence for an individual clause.

Example

The two "append" clauses

append(Nil,y,y)<-
append(u.x,y,u.z)<-append(x,y,z)

treat disjoint input tuples <Nil,y>, <u.x,y> for the appending use of the program. The two equivalences

append(Nil,y,z) <-> z=y
append(u.x,y,z) <-> ∃z'[z=u.z'&append(x,y,z')]

are therefore true of the append-relation computed by the program. Each can be derived from the equivalence

append(x,y,z) <-> x=Nil&z=y v
(∃u',x',z')[x=u'.x'&z=u'.z'&append(x',y,z')]

using the axioms

```
u.x≠Nil
u.x=u'.x'->u=u'&x=x'
```

of the identity theory. We shall give the derivation of the second equivalence.

Derivation

Instantiating the two clause equivalence we get

```
append(u.x,y,z) <-> u.x=Nil&y=z v
                    (∃u',x',z')[u.x=u'.x'&z=u'.z'&append(x',y,z')].
```

Using u.x≠Nil and u'.x'=u.x->u'=u&x'=x from the identity theory, together with the general substitutivity property of equality, the right hand side reduces to the equivalent

```
<-> (∃u',x',z')[u=u'&x=x'&z=u'.z'&append(x',y,z')].
```

Using the equivalence ∃x'(x=x'&W[x'])<->W[x] which holds for any formula W, this can be further simplified to

```
<-> ∃z'[z=u.z'&append(x,y,z')].
```

This is the equivalence we require.

Induction schemas

To use a set of equivalences as axioms for the program computed relations is to exploit the fact that we know that the interpretation comprising the set of program computed relations is a fixpoint of the program associated transformation **P**. We also know that it is the least fixpoint, that it gives each relation the smallest possible extension. If we make use of this fact, we can strengthen our axiomatisation with induction schemas.

Consider, once more, the append program. We know that the append relation computed by this program is the smallest set of triples of terms that includes all the triples of the form <Nil,t,t> and which, for any t, includes <t.t1,t2,t.t3> whenever it includes <t1,t2,t3>.

Now, suppose that W[x,y,z] is a formula with free variables x,y,z. For any Herbrand interpretation, W[x,y,z] will denote some ternary relation over terms. If this relation includes all the the triples of the form <Nil,t,t>, in other words if

```
(∀y)W[Nil,y,y]                                   (A)
```

is true, and if it includes <t.t1,t2,t.t3> whenever it includes <t1,t2,t3>, i.e. if

```
(∀u,x,y,z)(W[x,y,z]->W[u.x,y,u.z])   (B)
```

is true, then the relation denoted by W[x,y,z] includes all the triples

of terms in the append relation computed by the program. This is because this latter relation is the smallest relation with these properties. In other words,

```
(∀x,y,z)(append(x,y,z)->W[x,y,z])
```

is true of the append relation computed by the program and any W[x,y,z] for which (A) and (B) are true. Absorbing these conditions on W[x,y,z] in a single implication, we get the induction schema

```
for any W[x,y,z]
    (∀y)W[Nil,y,y] & (∀u,x,y,z)(W[x,y,z]->W[u.x,y,u.z]) ->
                                  (∀x,y,z)(append(x,y,z)->W[x,y,z]).
```

More generally, suppose that the set of clauses {C1,..,Ck} about an n-ary predicate R embody a recursive description of the R-computed relation in terms of some more primitive program computed relations R1,..,Rm. That is, each of the other predicates R1,..,Rm appearing in the antecedents of the clauses C1,..,Ck are described by a set of clauses that do not refer to the predicate R[1].

Let ∀C'i denote the universal closure of the clause Ci in which each occurrence of the predicate R has been replaced by the formula variable W. The induction schema

```
for any W[x1,..,xn]
   ∀C'1 & ..& ∀C'k -> (∀x1,..,xn)(R(x1,..,xn)->W[x1,..,xn])
```

is true of the program computed relations to which it refers. It is true because the R-computed relation has the least extension that can be assigned to the predicate R such that C1,..,Ck are true of the program computed relations R1,..,Rm.

Note that this axiom is just the Scott computation induction rule[deBakker & Scott 1969] expressed as a first order induction schema. Together with the program clauses for the predicate R, it gives us the strongest axiomatisation of the R-relation computed by the program that we can express in first order logic. The equivalence implicit in the set of clauses C1,..,Ck can be derived as a theorem.

---

[1] This excludes the case where the set of clauses about R are part of a mutually recursive description of several relations. Although the induction schema is still true of a relation given such a mutually recursive description, it is not the strongest induction schema that we can give. The strongest schema replaces each predicate involved in the mutual recursion with a formula variable.

4.3 Some example verifications

example-1

The appending use of the append program terminates if the sentence

$$(\forall x,y)[list(x)\&list(y) \rightarrow \exists z append(x,y,z)]$$

is true for "append" the relation computed by the program and "list" the relation comprising the set of terms

$$\{t1.—.tn.Nil : n\geq 0\}.$$

As our axioms for append we can use any axiom of the theory of this program computed relation. To these we must add axioms describing the list term relation.

The set of list terms is the smallest set of terms that includes Nil and which includes a term of the form t.t' whenever it includes the term t'. In other words, it is the relation computed by the logic program

$$list(Nil)\leftarrow$$
$$list(u.x)\leftarrow list(x) \qquad (A).$$

It follows that these clauses, and the equivalences

$$list(x) \leftrightarrow x=Nil \vee (\exists u,x')[x=u.x'\&list(x')]$$
$$list(u.x) \leftrightarrow list(x) \qquad (B),$$

and the induction schema

$$\text{for any } W[x]$$
$$W[Nil] \;\&\; (\forall u,x)[W[x]\rightarrow W[u.x]] \rightarrow \forall x[list(x)\rightarrow W[x]] \qquad (C),$$

can all be used as axioms for the relation.

Note that the two program clauses (A), the induction schema (C), and the two axioms

$$u.x\neq Nil$$
$$u.x=u'.x'\rightarrow u=u'\&x=x'$$

of the identity theory, are the analogues of the Peano axioms for the natural numbers. They are the axioms for the list-term data structure in our little theory of the relation over list terms computed by the append program.

Proof of termination

To prove the termination sentence we use the clauses of the append program and the list induction schema with W[x] as

$$\forall y[list(y)\rightarrow\exists z append(x,y,z)].$$

The proof is an induction over the input relation.

Proof of correctness

To show that the concatenation use is correct for the output relation

$$O=\{\langle t1,t2,t3\rangle : t3 \text{ a list term}\},$$

we must prove the correctness sentence

$$(\forall x,y,z)[list(x)\&list(y)\&append(x,y,z)\rightarrow list(z)].$$

Since this can be reformulated as either

$$(\forall x,y,z)[list(x)\rightarrow[list(y)\&append(x,y,z)\rightarrow list(z)]] \qquad (D)$$

or

$$(\forall x,y,z)[append(x,y,z)\rightarrow[list(x)\&list(y)\rightarrow list(z)]] \qquad (E),$$

we can use either the list induction schema or the append induction schema to prove correctness.

Method 1

We prove (D) using the append equivalences

$$append(Nil,y,z)\leftrightarrow z=y$$
$$append(u.x,y,z)\leftrightarrow\exists z'[z=u.z'\&append(x,y,z')],$$

and the list induction schema with W[x] as

$$\forall y\forall z[list(y)\&append(x,y,z)\rightarrow list(z)].$$

We give the proof as an illustration.

Basis: W[Nil]

We need to prove

$$\forall y\forall z[list(y) \;\&\; append(Nil,y,z) \rightarrow list(z)].$$

Assume list(y) and append(Nil,y,z) for arbitrary y and z. Using the "append" equivalence

$$append(Nil,y,z) \leftrightarrow y=z$$

we can infer y=z. With list(y) this implies the list(z) that we need.

Induction step: (∀u,x)(W[x] → W[u.x])

Assume

$$W[x]: \forall y'\forall z'[list(y') \;\&\; append(x,y',z') \rightarrow list(z')] \text{ (ind. hypothesis)}$$

and the antecedents

$$list(y),\ append(u.x,y,z),$$

of W[u.x] for arbitrary x,u,y and z. We try to derive the conclusion

list(z) of W[u.x].

Using append(u.x,y,z) and the equivalence

append(u.x,y,z) <-> ∃z'[z=u.z'&append(x,y,z')],

we can infer

z=u.z' and append(x,y,z') are both true

for some particular z. Using append(x,y,z'), assumption list(y), and the induction hypothesis, we can infer list(z'). This, together with z=u.z' and "list" axiom

list(u.x)<-list(x),

imply the list(z) that we require.

Method 2

We prove (E) using

list(Nil)
list(u.x)<->list(x),

and the append induction schema

∀yW[Nil,y,y] & (∀u,x,y,z)[W[x,y,z]->W[u.x,y,u.z] ->
(∀x,y,z)[append(x,y,z)->W[x,y,z]]

with W[x,y,z] as

[list(x)&list(y)->list(z)].

The structure of the proof is similar to that for method 1.

Method 1 is an induction over the input relation, and method 2 a computational induction on the append program. Note that in this case the induction over the input relation is also a computational induction over the "list" logic program that computes the relation.

example-2

Consider the logic program

perm(Nil,Nil)<-
perm(u.x,v.z)<-delete(v,u.x,y)&perm(y,z)

delete(u,u.x,x)<-
delete(v,u.x,u.y)<-delete(v,x,y)

and its use <perm(x1,y1),list(x1)> for permuting list terms. Let us try to prove that this is totally correct with respect to the output relation

PERM = {<t1.—.tn.Nil,t'1.—.t'm.Nil> : n=m &
each t'i=tj for some j
& each tj=t'i for some i}.

To do this, we must prove the total correctness sentence

∀x∀y[list(x)->(perm(x,y)<->PERM(x,y))].

Axiomatising the input relation

Firstly, let us address the question of the most suitable axiomatisation of the input relation. The most straightforward axiomatisation is the one that we used for example-1. However, 'looking ahead', we can see that the proof of the total correctness sentence using the induction schema

W[Nil] & ∀u∀x(W[x]->W[u.x]) -> ∀z(list(z)->W[z]),

will require us to show that

∀y[perm(x,y)<-> PERM(x,y)] (induction hypothesis)

implies

∀y[perm(u.x,y)<->PERM(u.x,y)].

To prove this implication we would have to use the equivalence

perm(u.x,w)<->(∃v,z,y)[w=v.z&delete(v,u.x,y)&perm(y,z)]

implicit in the perm program. However, this relates the permutation of u.x not to the permutation of x, which would enable us to use the induction hypothesis, but to the permutation of some y related to u.x by the program computed delete relation. This y is not a simple substructure of u.x. However, it is 'smaller'. It contains one less element. We could use an induction hypothesis which referred to smaller lists. We therefore need an inductive characterisation of the set of list terms based on length instead of structure.

What we have here is an example of a logic program that recursively computes an output for each term in some input relation I by:

(1) having one or more non-recursive clauses which together cover all the inputs that are bottom elements of I under some well-founded ordering <<.

(2) having one or more recursive clauses which together cover all the non-bottom elements of I, and which have recursive calls in which the input argument is reduced relative to <<.

In our case the well-founded ordering is the ordering of all the list terms based on the number of elements. We digress, temporarily, to discuss well-founded orderings and their associated induction schemas.

Well-founded orderings

DEFINITION
<< is a **well-founded ordering** of some set I if it is an irreflexive and transitive relation over I such that every non-empty subset A of I contains a bottom element. e is a **bottom element** of A if there is no e' in A such that e'<<e.

For each well-founded ordering << of a set I there is an induction principle.

Induction principle

If a set I' is such that it includes any e in I whenever it includes all the e'<<e, then I' includes all the elements in I.

Proof

Suppose not. Then the subset A of I that contains all the elements in I not in I' is non-empty. Since << is well-founded, this subset must contain a bottom element e. Because it is a bottom element of A, all the e'<<e must be in I'. In which case, by our assumption concerning the relationship between I and I', e must be in I' and not in A. This is a contradiction. I' must include all of I.

Formulation as an induction schema

Let W[x] be any formula with free variables x, and let I be some (input) relation over terms. Relative to any Herbrand interpretation, W[x] will denote a set I' of tuples of terms. Now suppose that << is a well-founded ordering of the set of tuples of terms that comprise the relation I. The above induction principle tells us that

```
for any W[x]
  ∀x(I(x) & ∀y[y<<x->W(y)]->W[x])
                    ->∀z(I(z)->W[z])
```

is a true statement about I and this well-founded ordering <<. Note that this is just Burstall's structural induction principle [Burstall 1969] expressed as a first order schema.

example-2 (continued)

Returning to our example, the relation

```
<< = {<t1.--.tm.Nil,t'1.--.t'n.Nil>: m<n}
```

is a well-founded ordering of the input relation of all the list terms. Hence,

```
for any W[x],
   ∀x[list(x) & ∀y(y<<x->W[y]) -> W[x]) -> ∀z(list(z)->W[z])
```

is true of the list input relation and this ordering of its extension. We can use it as an extra axiom about list terms.

To be of any use, we must also have some axioms describing the order relation. We shall use:

```
x<<Nil <-> false
```

to tell us that Nil is a bottom of all the list terms, and

```
y<<u.x <-> y=Nil v (∃v,y')[y=v.y'&y'<<x]
```

to tell us which list terms are less that some non-bottom list u.x.

The axiom x<<Nil<->false, and the equivalence

```
list(x) <-> x=Nil v (∃u,x')[x=u.x'&list(x)],
```

enable us to transform the new induction schema to

```
for any W[x],
 W[Nil] & ∀u∀x[list(x)&∀y(y<<u.x->W[y])->W[u.x]] ->
                                          ∀z(list(z)->W[z]).
```

This is the form that we shall use to prove the total correctness sentence.

Axiomatising the output relation

This takes care of the axiomatisation of the input relation. Let us now address the problem of axiomatising the relation

```
PERM = {<t1.—.tm.Nil,t'1.—.t'n.Nil> : n=m & each
                                        each t'i=tj for some j &
                                        each tj=t'i for some i}.
```

To do this, we must translate the above set theoretic definition into a set of equivalences.

First of all, the condition n=m is satisfied by two list terms

```
x=t1.--.tm.Nil, y=t'1.--.t'n.Nil
```

if and only if

$$\forall p[length(x,p) <-> length(y,p)]$$

is true of x and y. "length" is the relation

$$length = \{<t1.--.tn.Nil, s^n(0)> : n \geq 0\}.$$

Similarly, the condition

each t'i=tj for some j, each tj=t'i for some i,

is satisfied by x and y iff

$$\forall u[on(u,x) <-> on(u,y)]$$

is true. "on" is the relation

$$on = \{<t,t1.—.tn.Nil> : t=ti \text{ for some } i\}.$$

Hence, as our top-level definition of the PERM relation, we can use

$$PERM(x,y) <-> list(x) \,\&\, list(y) \,\&\, \forall p[length(x,p) <-> length(y,p)] \,\&\, \forall u[on(u,x) <-> on(u,y)].$$

We add to this axioms for the length and on relations. We shall use:

$$length(Nil,0)$$
$$length(u.x,s(n)) <-> length(x,n)$$

$$on(u,Nil) <-> false$$
$$on(u,v.x) <-> u=v \vee on(u,x).$$

Proof of total correctness

We shall simply sketch the proof. We use our new list induction schema with W[x] as

$$\forall y[perm(x,y) <-> PERM(x,y)].$$

Remember that we can use any axiom of the theory of the program computed relations.

The base case is the proof of

$$\forall y[perm(Nil,y) <-> PERM(Nil,y)] \qquad (A).$$

The induction step is the proof that

$$list(x) \,\&\, \forall w[w<<u.x \rightarrow \forall y[perm(w,y) <-> PERM(w,y)]$$

implies (B)

$$\forall y[perm(u.x,y) <-> PERM(u.x,y)].$$

(A) is easily proved using the axioms for perm and PERM. To prove (B) we need to use the following lemmas:

$$(\forall u,x,v,z)[list(x) \rightarrow [PERM(u.x,v.z) <-> \exists y(delete(v,u.x,y) \,\&\, PERM(y,z))]] \qquad (C)$$

$$(\forall v,x,y)[delete(v,x,y) \rightarrow y<<x] \qquad (D).$$

(C) is a 'qualified' total correctness result for the "delete" program. It can be proved using the simple induction schema for lists. (D) can be quickly proved using the "delete" induction schema implicit in its pair of clauses.

4.4 Proving other properties

We have used the theory of the program computed relations to prove correctness, termination and total correctness. We are not restricted to proving just these properties of programs. We can use the theory of the computed relations to prove any property that can be expressed by a sentence of first order logic.

For example, suppose that we want to prove that all calls of the "append" program of the form

$$<-append(t,Nil,z), \text{ t any term}$$

will, if they successfully terminate, result in z being bound to t. This will be the case if the sentence

$$\forall x \forall y[append(x,y,z) \rightarrow (y=Nil \rightarrow z=x)]$$

is true of the program computed relation.

Using the induction schema implicit in the append program, the proof of this sentence is very straightforward. We use the schema with W[x,y,z] as $x=Nil \rightarrow z=y$.

Functionality

Another property of the append program that is of interest is the fact that it is a function on its first two arguments. The sentence

$$(\forall x,y,z,w)[append(x,y,z) \,\&\, append(x,y,w) \rightarrow z=w].$$

expresses this fact. To prove it, we use the equivalences

$$append(Nil,y,z) <-> y=z$$
$$append(u.x,y,z) <-> \exists z'[z=u.z' \,\&\, append(x,y,z)],$$

the identity theory axiom

$$u.x=u'.x' \rightarrow u=u' \,\&\, x=x',$$

and the "append" induction schema with W[x,y,z] as

$$\forall w[append(x,y,w)\rightarrow y=z].$$

Associativity

The computed relation should also be associative. If we want to append three list terms t1,t2,t3 it should not matter if we first append t1 and t2, and then append the result to t3, or, if we first append t2 to t3, and then append t1 to the result. The associativity property holds if

$$(\forall x,y,z,w,s,t)[append(x,y,s)\&append(s,z,w)\&append(y,z,t)\rightarrow append(x,t,w)]$$

is true. To prove this we can again use the "append" induction schema. We need to appeal to the above functionality result.

Indirect proofs of program properties

Using the induction schema implicit in the program is the direct method of proving these 'algebraic' properties of a program computed relation. We can sometimes give indirect proofs of these properties by first proving some total correctness result. We then prove that the output relation has the properties in which we are interested.

As an example of this approach, let us see how we might use the total correctness theorem

$$\forall x[list(x)\rightarrow[perm(x,y)\leftrightarrow PERM(x,y)]]$$

for the program computed perm relation. Because of this conditional equivalence, it follows that any theorem we can prove about PERM applies to the perm relation when its first argument is a list term. It actually applies without this restriction. This is because

$$(\forall x,y)[perm(x,y)\rightarrow list(x)]$$

is a theorem that we can prove using the "perm" and "delete" induction schemas implicit in their respective clauses.

We leave the reader to check that each of the following are easily proved theorems about the PERM relation:

$$\forall x PERM(x,x)$$
$$\forall x\forall y[PERM(x,y)\rightarrow PERM(y,x]$$
$$\forall x\forall y\forall z[PERM(x,y)\&PERM(y,z)\rightarrow PERM(x,z)]\ .$$

Because PERM has an explicit definition, we do not need to use an induction. It follows that the same theorems hold when "PERM" is replaced by "perm", i.e. that the perm computed relation is reflexive, symmetric and transitive.

4.5 Consequence verification

Let us look again at the problem of showing that some use $\langle R(x\hat{}y),I\rangle$ is correct for an output relation O. We have to show that

$$(\forall x,y)[R(x\hat{}y)\rightarrow[I(x)\rightarrow O(x,y)]]$$

is true of the program computed relation.

In section 4.3 we tackled this problem by deriving the sentence as a theorem of the theory of the program computed relations. There is another method. By Theorem 3.8, we know that any atom $R(t\hat{}t')$ in the program computed relation denotes an instance of the relation assigned to "R" in any model of the program. Suppose that we can show that the Herbrand interpretation in which "R" is assigned the relation

$$\{\langle t\hat{}t'\rangle : t \text{ and } t' \text{ any term tuples, but if } t \text{ is in } I \text{ then } \langle t,t'\rangle \text{ is in } O\}$$

is a model of the program. It follows that if $R(t\hat{}t')$ is in the program computed relation, then

$$[I(t)\rightarrow O(t,t')]$$

is true. In other words, it follows that the correctness sentence is true.

To show that the interpretation in which "R" is assigned the above relation is a model of the program we can again make use of the truth preserving property of first order inference. We show that each of the program clauses are theorems about the relation characterised by the definition

$$R(x\hat{}y) \leftrightarrow [I(x) \rightarrow O(x,y)]$$

and a set of axioms for the relations I and O.

Programs as theorems

This indirect method for proving that a correctness sentence is true is just an application of a more general method for proving that the atoms of some program computed relation only denote instances of some particular relation R. As depicted in Fig. 4.2, we prove this by showing that each of the program clauses are theorems of the theory of an interpretation I in which "R" names the relation R. The interpretation I need not be a Herbrand interpretation.

Note the contrast with our previous method of proving correctness. There, the program clauses were used as axioms; now they are derived as theorems. Because the program clauses are consequences rather than assumptions, we call this method consequence verification. Its justification rests on the fact that each atom $R(t1,..,tn)$ of the R-relation computed by the program is, indirectly, a logical consequence of the axioms of the theory. It follows that the universal closure of $R(t1,..,tn)$ is true for any model of the axioms. By assumption, I is a model of the axioms. Hence, for interpretation I, $\langle t1,..,tn\rangle$ denotes a

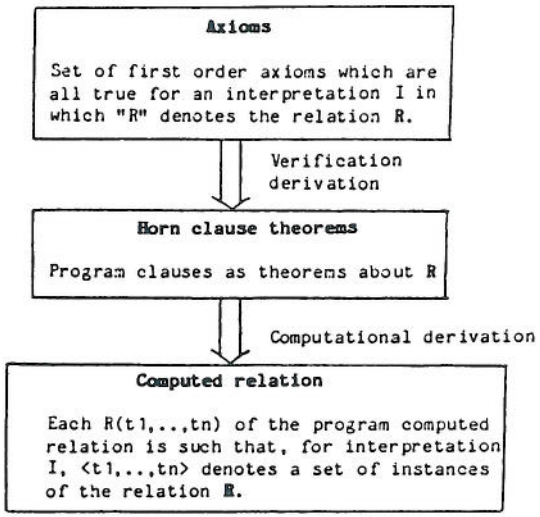


Fig. 4.2

set of instances of the relation R.

Relation to fixpoint induction

Fixpoint induction[Park 1969] is the proof rule

$P(I) \leq I$ implies $I_0 \leq I$, I_0 the least fixpoint of P.

It is a direct application of Tarski's fixpoint theorem. By Theorem 3.5, the antecedent of this rule is the condition that I is a Herbrand model of P. So we can re-express the rule

I is a Herbrand model of P implies each R(t1,..,tn) computed by P is true for I.

Consequence verification is the generalisation of this proof rule to

I a model of P implies universal closure of each R(t1,..,tn) computed by P is true of I.

Although consequence verification has a direct justification in terms of the model theory of first order inference, it can also be derived from the fixpoint rule by appealing to the model preserving property of the mapping H.

Applications of consequence verification

example-1

The set of sentences

```
list(Nil)
list(u.x) <-> list(x)

append(x,y,z) <-> [list(x)&list(y) <-> list(z)]
```

are true of the Herbrand interpretation in which "list" denotes the set of list terms and "append" denotes the relation

```
{<t1,t2,t3> : t1,t2,t3 any terms, but t3 is a list term
              if t1 & t2 are, and conversely t1 and t2
              are list terms if t3 is}.
```

If we can derive each of the program clauses

```
append(Nil,y,y)<-
append(u.x,y,u.z)<-append(x,y,z)
```

as theorems, we will prove that the appending use of this program, and the decomposition use, both map list terms into list terms.

Derivations

We instantiate the append definition for the tuple of terms treated by each program clause and try to reduce the definiens to the antecedent of the clause.

```
append(Nil,y,y) <-> [list(Nil)&list(y)<->list(y)]
                    using axiom list(Nil)          (1)
                <-> [list(y)<->list(y)]
                                                   (2)
                <-> true.
```

So, append(Nil,y,y) is a theorem.

```
append(u.x,y,u.z) <-> [(list(u.x)&list(y)<->list(u.z)]
                      using list(u.x)<->list(x)    (3)
                  <-> [(list(x)&list(y)<->list(z)]
                                                   (4)
                  <-  append(x,y,z).
```

Hence,

```
append(u.x,y,u.z)<-append(x,y,z)
```

is a theorem.

Note that the proof steps (1),(2),(3) and (4) are just the proof steps that we would need to prove

(∀x,y,z)[append(x,y,z) -> [list(x)&list(y)<->list(z)]]

using the append induction schema. The only difference is the meta-theory that justifies the claim that the display of these proof steps proves something about the "append" program.

example-2

The sentences

append(x,y,z) <-> length(z)=length(x)+length(y)

length(Nil)=0
length(u.x)=s(length(x))

0+n=n
s(m)+n=s(m+n)

are all true of an interpretation in which:

"length" denotes a function which maps a list term t1.--.tn.Nil into the number n,

"+" is the number addition function,

"0" is the number zero,

"s" is the function which adds 1 to any number,

"append" is a subset of the set of triples of list terms that includes <t1,t2,t3> only if the length of t3 is the sum of the lengths of t1 and t2.

By deriving each "append" program clause as a theorem of this append relation, we show that the relation it computes has this length addition property for list term arguments.

example-3

The equivalence

append(x,y,z) <-> [y=Nil->x=z]

is true of the relation over the Herbrand universe which includes any triple of terms <t1,t2,t3> for which t1≠Nil. However, it includes a triple of the form <t1,Nil,t3> only if t1=t3.

By deriving the "append" program clauses as theorems, we prove that the computed relation is a subset of the relation defined by this equivalence. So the computed relation is also restricted by the condition that append(t1,Nil,t3) is true only if t1=t3.

4.6 Notes and references

The method of verifying a Horn clause program by proving theorems in the theory of the relations it computes was first investigated in [Clark & Tarnlund 1977]. It is similar to the method used by McCarthy[1977] and Cartwright[1977] to verify LISP programs, and by Burstall[1969] to verify programs in an ALGOL subset. In each case, theorems are proved about the program computed relations in a first order theory of these relations. The advantage that logic programs have, is that the mapping from program to axioms of the theory is trivial.

The consequence verification method for verifying logic programs was first investigated in [Clark 1977].

Chapter 5 Deductive program construction

At the end of the last chapter we investigated a method of verifying logic programs that involved deriving each clause as a theorem of the theory of the relations it was supposed to compute. We can think of the axioms of this theory as the specification of the program. Suppose that we start with the specification rather than the program, and suppose that by systematically deriving computationally useful Horn clause theorems we manage to piece together a logic program. This is a deductive program construction.

In this chapter we look at this deductive approach to the construction of a logic program. First, we describe and exemplify a method for systematically searching for a set of Horn clause theorems about a relation. This method is an extension of the Darlington and Burstall techniques [Darlington 1975, Bustall & Darlington 1975], for deriving programs specified and written in recursion equations, to the domain of programs specified in predicate logic and written in the Horn clause subset. We then show how we can sometimes piece together several different programs for a given relation by combining different theorems of the same specification. Finally, we show how we can apply our program derivation methods to the task of transforming a given program into a program with a quite different recursive structure. We give two examples of this. In each case we transform a given program P into a program P' such that a strictly sequential execution of P' emulates a coroutining execution of P.

5.1 Searching for Horn clause theorems

We want to find a set of Horn clauses that can be used to compute some subset of a relation R which is large enough for us to be able to prove that the program terminates for one or more intended uses. Using the full expressive power of first order logic, we can invariably find some formula W[x1,..,xk] which, for some interpretation, constitutes an intuitively correct description of the relation. We can then specify the relation using the definition

R(x1,..,xk) <-> W[x1,..,xk]

and a set of axioms for the relations and functions referred to by the definiens W[x1,..,xk].

We know that each clause of a logic program for R will be an implication with a consequent atom R(t1,..,tk). The tuple of terms <t1,..,tk> will denote some candidate set of instances of the relation and the antecedent of the clause will expresses some condition on t1,..,tk which ensures that it only denotes instances of the relation R.

Let <t1,..,tk> be a tuple of terms for which it is reasonable to assume there are instances that fall inside the specified relation. The definition instance

R(t1,..,tk) <-> W[t1,..,tk]

gives us exactly the condition that the tuple of terms must satisfy if they are to denote only instances of R. If we could 'reduce' this condition to a conjunction of atoms, we would have derived a Horn clause theorem.

We can try to reduce it to a conjunction of atoms by engaging a step-by-step transformation of the definiens W[t1,..,tk]. At each step we use a specification axiom or a logical manipulation to produce a new formulation of W[t1,..,tk] which, if not equivalent to the previous formulation, at least implies it. This implication constraint ensures that if we do manage to reduce W[t1,..,tk] to a conjuction of atoms A1&..&Am, then

R(t1,..,tk) <- A1&..&Am

is a Horn clause theorem.

Fold substitutions

One transformation step is of special significance. It is the one which enables us to derive recursive clauses. Suppose that we have managed to reduce W[t1,..,tk] to a formula of the form

..&W[t'1,..,t'k]&..

in which a new instance of W[x1,..,xk] appears as a sub-formula. By our definition of R, W[t'1,..,t'k] is equivalent to R(t'1,..,t'k). We can therefore transform

..&W[t'1,..,t'k]&..

into

..&R(t'1,..,t'k)&..

and in so doing derive a recursive implication

R(t1,..,tk) <- ..&R(t'1,..,t'k)&..

'Fold substitution' is the name given to this transformation step by Darlington and Burstall. Trying to factor out the W[t'1,..,t'k] in order to apply the fold substitution will be a major goal of the transformation for any <t1,..,tk> for which we can expect to find a recursive theorem. As a method for deriving recursive programs from their specifications, it was developed, independently, by Darlington and Burstall[1975] and by Manna and Waldinger[1975].

5.2 An example program derivation

Let us consider the problem of constructing a program to compute the min relation, the relation that holds between a list of elements x and an element u when u is the least element on x under some total order relation $\leq$ for elements. Although the derivation of a program to compute this relation is relatively straightforward, the task is complex enough for us to be able to illustrate several of the most useful derivation strategies.

Specification

We must find a set of axioms that constitute an intuitively correct description of the min relation. As our top-level definition, we shall say that u is the minimum of x iff u is on x and it is a lowerbound of x. Formalising this gives us the equivalence

min(u,x) <-> on(u,x)&lowerbound(u,x) (A1).

Something u is a lowerbound of x iff it is less than or equal to everything on x.

lowerbound(u,x) <-> ∀w[on(w,x)->u≤w] (A2).

Nothing is on the empty list Nil, and something is on a constructed list v.x iff it is v or it is on x.

on(u,Nil) <-> false (A3).

on(u,v.x) <-> u=v ∨ on(u,x) (A4).

That $\leq$ is a total order relation over some set of objects called elements, is specified by the set of axioms:

element(u) -> u≤u (A5)

element(u)&element(v) -> u≤v ∨ v≤u (A6)

u≤v&v≤w -> u≤w (A7)

u≤v&v≤u -> u=v (A8)

u≤v -> element(u)&element(v) (A9).

We shall assume that there already are logic programs for the predicates "element" and "$\leq$". The axioms (A5)-(A9) express a set of properties of the relations computed by these programs that we can make use of in our derivation. These relations have the role of parameters of the specification.

Derivation

We know that any instance of the min relation must be named by a pair of terms that are a substitution instance of <u,v.x>. This is because there are only minimum elements of constructed lists. We shall therefore try to derive a simple recursive characterisation of some

subset of instances of this form by transforming the definiens of

min(u,v.x) <-> on(u,v.x)&lowerbound(u,v.x).

Using axiom (A4), we can expand this to

min(u,v.x) <-> [u=v ∨ on(u,x)] & lowerbound(u,v.x),

which is equivalent to

min(u,v.x) <-> u=v&lowerbound(u,v.x) ∨ on(u,x)&lowerbound(u,v.x).

Dropping the only-if, this gives us a pair of implications

min(u,v.x) <- u=v & lowerbound(u,v.x) (T1)

min(u,v.x) <- on(u,x) & lowerbound(u,v.x) (T2).

(T2) already has one of the conditions, on(u,x), that define min(u,x). If we could reduce lowerbound(u,v.x) to some conjuction involving lowerbound(u,x), we could apply a fold to get a recursive clause for min.

Using definition (A2), we can expand the antecedent of (T2) to give the equivalent implication

min(u,v.x) <- on(u,x) & ∀w[on(w,v.x)->u≤w].

Applying (A4) again, this becomes

min(u,v.x) <- on(u,x) & ∀w[(w=v ∨ on(w,x))->u≤w].

Using the logical laws

[A ∨ B -> C] <-> [A->C] & [A->C]

∀x[A&B] <-> ∀xA & ∀xB

this can be further transformed into

min(u,v.x) <- on(u,x) & ∀w[w=v->u≤w] & ∀w[on(w,x)->u≤w].

The formula ∀w[on(w,x)->u≤w] is the definiens of lowerbound(u,x) that we require. We can therefore fold using the lowerbound definition, and then fold again using the min definition. This gives us

min(u,v.x) <- ∀w[w=v->u≤w] & min(u,x).

Let us now look at the condition ∀w[w=v->u≤w]. It is equivalent to the condition u≤v. This is because, for any formula W[x], ∀x[x=u->W[x]] is equivalent to W[u]. Applying this last simplification gives us the Horn clause theorem

min(u,v.x) <- u≤v & min(u,x) (T3).

As a component of a logic program for the min relation, it would cover

the case where the minimum appeared on the tail of the list.

Let us see if we can effect a similar transformation of (T1). Expanding lowerbound(u,v.x) as in the derivation of (T3) will eventually give us

on(u,v.x) <- u=v & u≤v & ∀w[on(w,x)->u≤w].

In the context u=v, the condition u≤v can be simplified to element(v). This is because the conjunction u=v&u≤v is equivalent to the conjunction u=v&v≤v by the substitutivity property of equality. Axiom (A5) tells us that v≤v is implied by element(v). Trading u≤v for element(v), and folding using the definition of lowerbound, will give us

min(u,v.x) <- u=v & element(v) & lowerbound(u,x).

'Anding' in extra conditions

This time we have not managed to factor out the complete definiens of min(u,x), but we do have one component. Suppose that we add the extra condition on(u,x) to the antecedent. This strengthening of the antecedent by 'anding' in extra conditions is a legitimate transformation step. It produces a new antecendent that implies the previous antecedent as required. Moreover, strengthening, in order to apply a fold substitution, often leads to a computationally useful recursive description. Let us see what it produces in this case.

min(u,v.x) <- u=v&element(v)&on(u,x)&lowerbound(u,x) (stengthening)

min(u,v.x) <- u=v&element(v)&min(u,x) (folding)

This clause could be used as part of a logic program for min. Its major drawback is that it would only succeed, when used to find the minimum element, if the minimum is the head of the list and it reappears on the tail of the list. We would be better served by a recursive theorem which did not insist that u appeared on x. Let us go back to

min(u,v.x) <- u=v&element(v)&lowerbound(u,x)

and try again.

Generalising the antecedent

In order to introduce a recursive call in which the element argument is not u we must first generalise the condition lowerbound(u,x) to lowerbound(w,x), w some new variable. Unfortunately, if we just replace u by w, the new antecedent

u=v&element(v)&lowerbound(w,x)

will not imply the previous antecedent

u=v&element(v)&lowerbound(u,x).

This flaunts our implication constraint. We can only generalise lowerbound(u,x) to lowerbound(w,x) if we simultaneously introduce

another condition R(u,w) such that

R(u,w)&lowerbound(w,x) -> lowerbound(u,x).

This generalisation subject to a constraint is another very useful transformation step. In this case, the extra condition we need is u≤w. The implication

u≤w&lowerbound(w,x) -> lowerbound(u,x)

is clearly true of the lowerbound relation of our intended interpretation. We could assume it as an extra axiom. However, this would be redundant. It can be derived as a lemma using the lowerbound definition and the transitivity axiom for ≤. Using this implication to justify our transformation, we can generalise as required. This gives us

min(u,v.x) <- u=v&v≤w&lowerbound(w,x).

Notice that we have also dropped the condition element(v). This is because, by axiom (A9), v≤w implies it.

We can now try our stengthening strategy again. We can 'and' in the condition on(w,x), and then fold, to give

min(u,v.x) <- u=v&v≤w&min(w,x) (T4).

Our two Horn clause theorems (T3) and (T4) together cover the two cases:

(1) minimum is the front of the list
(2) minimum is on the tail of the list.

To make them into a terminating program we need some base case non-recursive clause. The obvious base case is unit lists. To complete the program we need the clause

min(u,u.Nil) <- element(u).

We leave the reader to check that a transformation of

min(u,u.Nil) <-> on(u,u.Nil)&lowerbound(u,u.Nil)

quickly confirms that this is a theorem. Alternatively, we could assume it as a redundant axiom, since it is obviously true of the min relation we have tried to specify. The precondition, element(u), restricts the unit list instances of the min relation to lists of elements, which is what we intended.

Bringing together our Horn clause theorems, we have

min(u,u.Nil) <- element(u)

min(u,v.x) <- u≤v & min(u,x)

min(u,v.x) <- u=v & v≤w & min(w,x).

Termination

The above set of clauses are necessarily a correct program for computing the min relation as specified. As a logic program, they can be used for finding the minimum of a list of elements, confirming that some element is a minimum of such a list, even for generating lists on which a given element appears as the minimum. The usual use will be to find a minimum. Using the induction schema

```
for any W[x]
 (element(u)->W[u.Nil]) & ∀u(element(u)&W[x]->W[u.x])
                          ->∀x(list-of-els(x)->W[x])
```

as the axiom characterising non-empty lists of elements, we can easily prove that the use <min(y,x),list-of-els(x)> terminates. In fact, for this use, we can drop the condition element(u) of the base case clause. It is implied by the input relation. Thus, the set of clauses

```
min(u,u.Nil) <-

min(u,v.x) <- u≤v & min(u,x)

min(u,v.x) <- u=v & v≤w & min(w,x)
```

are a correct, terminating program for finding the minimum of a non-empty list of elements.

Conversion into a logic algorithm

Let us now address their computational use as a PROLOG program to find minimums. We need to rearrange the antecedent atoms of the recursive clauses to specify the best computation rule control. However, there is a more serious problem. Each recursive clause always computes the minimum of the tail of the list, but, in general, only one of these clauses will succeed. No matter how we order the clauses, there will be cases when the minimum of the tail is computed redundantly. To avoid this redundancy, we must find a single clause that subsumes the two clauses.

Fortunately, this is not too difficult to achieve. The clause

```
min(u,v.x) <- u≤v & min(u,x)
```

is logically equivalent to

```
min(u,v.x) <- u=w & w≤v & min(w,x).
```

This means that the pair of recursive clauses are equivalent to the non-clausal implication

```
min(u,v.x) <- (u=w&w≤v ∨ u=v&v≤w) & min(w,x)
```

To reduce this to a clause, we factor out the disjunction making it the

definiens of a new relation smaller(u,v,w):

```
smaller(u,v,w) <-> u=w&w≤v ∨ u=v&v≤w.
```

Definition introduction, in order to obtain a Horn clause implication, is another standard derivation strategy. It corresponds to the decision to compute some condition using an auxiliary logic program to be derived by transforming the introduced definition.

In this case the derivation of the auxiliary program is trivial. We just drop the only-if, and expand the resulting implication onto the pair of clauses

```
smaller(v,v,w) <- v≤w

smaller(w,v,w) <- w≤v,
```

The relation computed by these clauses is exactly the defined relation.

Our final logic algorithm is expressed by the list of clauses:

```
min(u,u.Nil) <-

min(u,v.x) <- min(w,x) & smaller(u,v,w)

smaller(v,v,w) <- v≤w

smaller(w,v,w) <- w≤v
```

together with the PROLOG control implicit in their text order.

5.3 Different program=different set of theorems

From a single specification we can sometimes derive a redundant number of computationally useful theorems. When this happens, we can often piece together different programs by taking different subsets of these theorems.

An example of this is provided by the range of useful Horn clause theorems that we can derive from the following specification of the input/output relation of a logic program to manipulate ordered labelled trees.

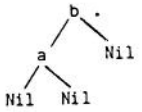
Specification

```
insert(x,u,y) <-> [ [ordered(x)<->ordered(y)] &
                    ∀v[label-of(v,y)<->(v=u ∨ label-of(v,x)]]

ordered(Nil)

ordered(t(x,v,y)) <-> ordered(x)&ordered(y)
                      &upperbound(v,x)&lowerbound(v,y)

lowerbound(v,x) <-> ∀w[label-of(w,x)->v<w]

upperbound(v,y) <-> ∀w[label-of(w,y)->w<v]

label-of(u,Nil) <-> false

label-of(u,t(x,v,y)) <-> u=v ∨ label-of(u,x) ∨ label-of(u,y)
```

axioms specifying that < is a relation which is a total strict order relation for a set of objects called labels

In the intended interpretation for these specification axioms the constant "Nil" denotes the tree with no labels and the functor "t" denotes a tree constructor that takes a labelled tree x, a label v, and a labelled tree y into a labelled tree t(x,v,y). So,

t(t(Nil,a,Nil),b,Nil) denotes the tree



The predicate "insert" names a relation that includes a labelled tree x, a label u, and a labelled tree y iff the label u and all the labels on x appear on y, but no other label appears on y, and, if either tree is ordered so is the other. The predicate "ordered" denotes this ordered set of labelled trees. A tree is ordered iff its root label is an upperbound of all the labels on its left subtree, and a lowerbound of all the labels on its right subtree, and its two subtrees are themselves ordered. Our upperbound and lowerbound definitions require a lowerbound of some labelled tree to be strictly less than every label on the tree, and an upperbound to be strictly greater than every label on the tree. Note that this means that an ordered tree only has one occurence of any label. Finally, the predicate "label-of" denotes the relation that includes a label u and a tree x iff u is some label on x, and the predicate "<" names the strict order relation for labels for which we assume there is some logic program.

The specification is such that any logic program comprising a set of theorems of the specification has the potential to be used both for inserting and deleting labels on ordered trees.

The insert use will be some goal clause of the form

<-insert(t,l,y), t a tree term, l a label not on t.

If t denotes an ordered tree, then any answer binding computed for y must also denote an ordered tree. Moreover, it will denote an ordered tree that is some reconfiguration of the input tree with the new label added.

The delete use is a goal clause of the form

<-insert(x,l,t), t a tree term, l a label on t.

Again, if t denotes an ordered tree, then any answer binding computed for x must denote an ordered tree which is some reconfiguration of the input tree with the label l deleted.

The specification imposes no structural constraints on the possible reconfigurations of the input tree for either use. If our derived program computes the full extension of the relation as specified, there will be many possible output bindings. For termination, we only need to be able to compute one output binding. What this means, is that we can get away with a logic program that computes quite a small subset of the specified relation.

We can restrict ourselves to such a subset, and at the same time impose some structural constraints on the reconfigurations of the input tree for either use, by only looking for Horn clause theorems with a consequent atom of the form insert(t,u,t') where t and t' share variables. Thus, an attempt to find a Horn clause theorem with consequent atom

insert(t(x,u,y),v,t(x,u,y'))

is an attempt to find a clause whose insert use will put the new label in the right subtree. An attempt to find a theorem with consequent atom

insert(t(x,u,y),v,t(t(x',u,y'),v,y"))

is an attempt to find a clause whose delete use will remove a root label and promote its left descendant to the root. It is reasonable to assume that there are instances of the insert relation that satisfy these structural constraints. What we must find is a recursive characterisation of such a restricted set of instances of the relation.

By making these sorts of guesses about the structural relations that might be satisfied by various subsets of the insert relation, and by transforming the corresponding instance of our insert definition, we can

derive each of the following theorems:

```
insert(t(x,u,y),v,t(x,u,y')) <- u<v&insert(y,v,y')                    (1)

insert(t(x,u,y),v,t(x',u,y)) <- v<u&insert(x,v,x')                    (2)

insert(t(x,u,y),v,t(t(x,u,y'),v,y")) <- u<v&insert(y,v,t(y',v,y"))    (3)

insert(t(x,u,y),v,t(x',v,t(x",u,y))) <- v<u&insert(x,v,t(x',v,x"))    (4)

insert(Nil,u,t(Nil,u,Nil)) <-                                         (5)
```

The derivations are more complex than those for the min program, but not unduly so.

Different programs

By taking different subsets of these theorems we get the following programs.

Program 1

Clauses (1),(2) and (5) constitute a terminating program for the uses:

<-insert(t,l,y), t a tree term, l a label not on t

<-insert(x,l,t), t a tree term, l a leaf label of t.

The insert use will generate an output binding identical to the input term except for the addition of the new label at the end of some branch, i.e. as a leaf label.

Program 2

Clauses (3),(4) and (5) constitute a terminating program for the uses:

<-insert(t,l,y), t a tree term, l a label not on t

<-insert(x,l,t), t a tree term, l the root label of t.

The insert use will generate an output binding in which the new label is the root of the tree.

Program 3

Clauses (1)-(5) constitute a terminating program for the use:

<-insert(x,l,t), t a tree term, l any label on t.

They will delete any label that appears on the input tree.

We leave the reader to check that they are terminating programs for each described use. For each program we know that the output binding must denote an ordered tree if the input term denotes an ordered tree.

5.3 Program transformation

The set of equivalences implicit in the clauses of a Horn clause program can be used as axioms describing the set of relations that the program computes. By treating these equivalences as specification axioms implicitly given by the program clauses, we can apply our transformation/fold techniques to try to derive a new set of clauses. Since these clauses will be theorems of the theory of the relations computed by the given program, the new program must compute a subset of each relation computed by that program. To prove that it computes exactly the extension of some particular relation, we need a separate inductive proof.

example-1

For our first example of this type of program transformation let us look again at the logic program

```
front(n,x,z)<-append(x,y,z)&length(x,n)

length(Nil,0)<-
length(u.x,s(n))<-length(x,n)                          (A)

append(Nil,x,x)<-
append(u.x,y,u.z)<-append(x,y,z)
```

that we considered in Chapters 1 and 2. This program implicitly provides us with the definition

front(n,x,z) <-> ∃y[append(x,y,z)&length(x,n)]

for the front relation that it computes. Let us see if we can find an alternative recursive description of the relation.

To do this, we apply our transformation/fold techniques to this definition. However, since we are dealing with a definition implicitly given by a program clause, we can indirectly transform the definition by evaluating the goal clause

<-front(n,x,z).

Each partial evaluation of this goal clause which reduces it to a derived goal clause <-C is the derivation of the Horn clause implication

[front(n,x,z)]s <- C, s the composition of unifiers of the evaluation path to C

from the program clauses. To try to find a recursive description of the front relation we should try to reduce <-front(n,x,z) to a goal clause of the form

<-..append(t,y',t')..length(t,t").. ,

where y' is a variable that does not appear in any other atom of the clause, nor in [front(n,x,z)]s. If we can do this, we have derived an

implication of the form

[front(n,x,z)]s <- ...∃y'[append(t,y',t')&length(t,t")]..

to which we can apply a fold substitution.

Fig. 5.1 is a partially constructed search tree for the evaluation of <-front(n,x,z) using the coroutining computation rule that alternates between the append and length sub-computations. The branch leading to

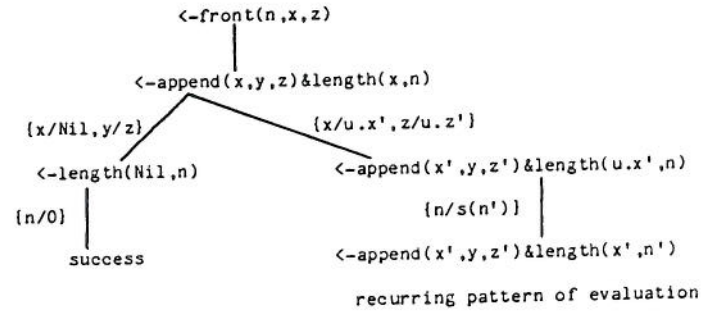


Fig. 5.1

the success node is the derivation of the assertion

front(0,Nil,z)<- .

That leading to the clause

<-append(x',y,z')&length(x',n')

is the derivation of

front(x,y,z){x/u.x',z/u.z',n/s(n')}<-
∃y[append(x'y,z')&length(x',n')].

Applying a fold substitution, we get the recursive clause

front(s(n'),u.x',u.z')<-front(n',x',z').

Completeness of the new program

The pair of clauses

front(0,Nil,z)<- (B)
front(s(n'),u.x,u.z')<-front(n',x',z')

are necessarily a correct program for computing instances of the front relation computed by our original program. This is because it comprises a pair of theorems of the theory of that relation. So the front

relation computed by the new program is included in the front relation computed by the original program. To prove the converse result, that the front relation computed by the new program (B) includes that computed by the original program (A), we can use either of two arguments.

1: Induction on the structure of the complete search tree

The two clauses (B) were derived by exploring the construction of some search tree for the goal clause <-front(n,x,z). The partial construction that lead to their derivation actually displayed a recurring pattern in the construction of this search tree. The complete search tree generated by the coroutining computation rule used to construct Fig. 5.1 has the form depicted in Fig. 5.2. The set of

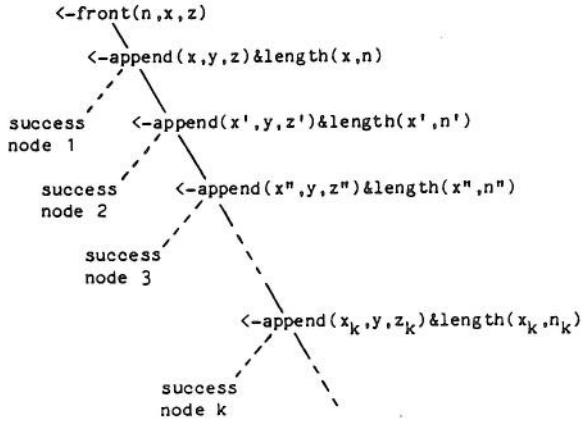


Fig. 5.2

answer substitutions given by the success terminating branches of this complete search tree are the set S of answer substitutions that can be computed for <-front(n,x,z) using this computation rule and program (A). Each of these answer substitutions is the answer given by the path ending in success node k, for some k. By an induction on k, we can prove that every answer substitution in S can be computed by evaluating the goal clause <-front(n,x,z) using program (B). Since the Herbrand extension of the set of answer S is the front relation computed by program (A) (Theorem 3.10), it follows that this relation is included in the front relation computed by the new program (B).

2: Induction in the theory of the computed relations

The front relation computed by program (A) is the relation

$$\exists y[\text{append}(x,y,z)\&\text{length}(x,n)].$$

To prove that the front relation computed by program (B) includes this relation, we can prove that

$$(\forall n,x,z)\{\neg y[\text{append}(x,y,z)\&\text{length}(x,n)]\rightarrow\text{front}(n,x,z)\} \quad (C)$$

is true of the relations computed by the set of clauses

```
append(Nil,y,y)<-
append(u.x,y,u.z)<-append(x,y,z)

length(Nil,0)<-
length(u.x,s(n))<-length(x,n)

front(0,Nil,z)<-
front(s(n),u.x,u.z)<-front(n,x,z).
```

To do this, we derive (C) is a theorem of the theory of these relations. Using the append induction schema implicit in its program clauses, and the equivalence axioms

```
length(Nil,n)<->n=0
length(u.x,n)<->∃m[n=s(m)&length(x,m)]
```

for the length relation, we can give an inductive proof of the equivalent sentence

$$(\forall n,x,y,z)[\text{append}(x,y,z)\rightarrow[\text{length}(x,n)\rightarrow\text{front}(n,x,z)]].$$

This proves that the front relation computed by (B) includes that computed by (A).

Comparison of algorithms

For the use <front(x1,y,x2),number(x1)&list(x2)>, the sequential execution of program (B) corresponds to the data flow coroutining execution of program (A) that we described in Chapter 2. This is because the recursive clause of (B) freezes the recurring pattern of this coroutining execution.

example-2

In the above example we were lucky. The recurring pattern of evaluation involved the conjunction of atoms that defined the relation for which we sought a recursive description. When this is not the case, we have to modify the given program by introducing some new relations.

An example of the need to do this is provided by the transformation of the

eight queens program:

```
Queen-sol(x,y)<-Perm(x,y) & Safe(y)

Perm(Nil,Nil)<-
Perm(u.x,v.z)<-delete(v,u.x,y) & Perm(y,z)

delete(u,u.x,x)<-
delete(v,u.x,u.y)<-delete(v,x,y)

Safe(x)<-Safe-pair(Nil,x)

Safe-pair(y,Nil)<-
Safe-pair(y,u.x)<-no-take(u,y,1) & Safe-pair(u.y,x)

no-take(u,Nil,n)<-
no-take(u,v.x,n)<-no-diagonal(u,v,n) & no-take(u,x,s(n))

no-diagonal(u,v,n)<-v>u & v=u+w & w≠n
no-diagonal(u,v,n)<-u>v & u=v+w & w≠n.
```

The recurring pattern of the coroutining execution of this program does not involve the Perm and Safe calls of the Queen-sol clause. It is a recurrence of Perm and Safe-pair calls. To capture this as a recursive description, we must first modify and then generalise the Horn clause description of the Queen-sol relation.

Clearly, a change of the above Queen-sol clause to

```
Queen-sol(x,y) <- Perm(x,y) & Safe-pair(Nil,y)
```

will not effect the relation computed by the program. We can then generalise this to

```
Queen-sol(x,y,z) <- Perm(x,y) & Safe-pair(z,y)  (1).
```

This makes the relation computed by the original program the relation Queen-sol(x,y,Nil). We need to generalise on the "Nil" in this way since this argument of the Safe-pair call is changed during the computation. We must therefore have a variable in this position if we are to have any chance of deriving a recursive description of the Queen-sol relation.

We can now apply the same derivation strategy as for the front program. We explore the evaluation of <-Queen-sol(x,y,z) using the computation rule that coroutines between the Perm and Safe-pair sub-computations. Fig. 5.3 depicts the partial construction of the corresponding search tree.

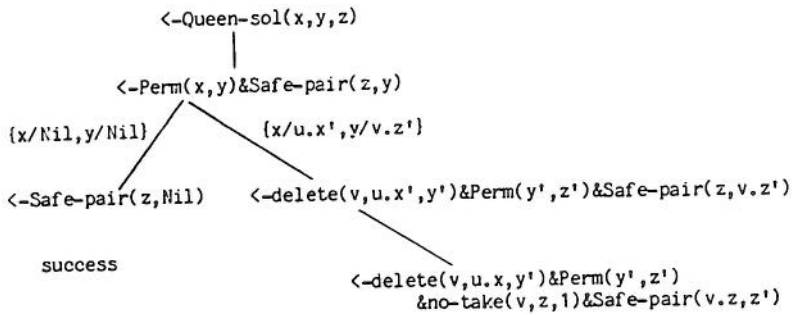


Fig. 5.3

The success branch records the derivation of the assertion

Queen-sol(Nil,Nil,z)<- (2)

The non-terminated branch records the derivation of the clause

Queen-sol(u.x',v.z',z) <- delete(v,u.x',y')&no-take(v,z,1)
&Perm(y',z')&Safe-pair(v.z.z').

Folding, using the Queen-sol equivalence implicit in clause (1), gives us the recursive clause

Queen-sol(u.x',v.z',z) <- delete(v,u.x',y')&no-take(v,z,1) (3)
&Queen-sol(y',z',v.z).

Completeness

Clauses (2),(3), together with the original program clauses for the delete, no-take and no-diagonal relations are a correct, complete alternative to the original program. Correctness again follows from the fact that (2) and (3) are theorems of the theory of the relations computed by the original program. To prove completeness, we can give an inductive proof that each of the answer substitutions computed by completed version of the search tree of Fig. 5.3. can also be computed using the new program. Alternatively, we can prove that

(Vx,y,z)[Perm(x,y)&Safe-pair(z,y) -> Queen-sol(x,y,z)]

is true of the Perm, Safe-pair and Queen-sol relations computed by the program comprising the clauses of the original program with (1) deleted and (2) and (3) added. This will be an inductive proof using either the "Perm" or "Safe-pair" induction schemas implicit in their respective program clauses.

Comparison of algorithms

For the evaluation of the goal clause

<-Queen-sol(1.2...8.Nil,y,Nil),

a strictly sequential execution of the new program corresponds to the data flow coroutining execution of the original program.

5.5 Notes and references

The application of the Darlington & Burstall transformation techniques to the problem of deriving Horn clause programs was first investigated, independently, by Clark and Sickel (Clark[1977], Clark & Sickel[1977]) and by Hogger[1977]. Hogger then went on to make a thorough study of this approach to the deductive construction of Horn clause programs, which he reported in his thesis[Hogger 1978]. This contains many examples of the derivation of programs. His approach is a little more formal than that presented here.

Bibel[1976,1978] has also investigated broadly similar methods for deriving logic programs, although his logic programs are not sets of Horn clauses. More recently, Hansson and Tarnlund[1979] have investigated a quite different method for deriving Horn clause programs. They do not make use of the transformation/fold method. They first guess an inductive structure for the program, and then derive the clauses that fit that structure using a natural deduction style inference system.

In [Clark & Darlington 1980] several different sort programs are derived from the same specification. Like the insert programs, they each comprise a different set of theorems. In this case the difference is not that they compute different subsets of the specified relation. It is that they embody different logical reformulations of the specification. These result from the use of different lemmas, and the use of alternative transformation steps, during the program derivation. These differences in the derivation history are used to classify and compare the (sequential control) algorithms implicit in the different sets of theorems.

Transformations of Horn clause programs, particularly transformations that can be proved to preserve equivalence of some program computed relation, have not been widely investigated. There is a weak analogy here with the transformation of a context free grammar that preserves equivalence of the language generated from a particular non-terminal. However, if there is anything to the analogy, there may be some normal form results for Horn clause programs that fall out of some general theorems concerning equivalence preserving transformations.

References

deBakker J.W. & Scott, D.[1969], A theory of programs, Unpublished notes, IBM seminar, Vienna.

Baxter,L.D.[1973], An efficient unification algorithm, Technical report CS-73-23, Faculty of Math., University of Waterloo.

Bibel,W.[1976], Synthesis of strategic definitions and their control, Report no. 7610, Tech. Univ. Munchen, FB Mathematic.

Bibel,W.[1978], On strategies for the synthesis of algorithms, Proceedings of AISB/GI Conference on Art.Int., Hamburg.

Boyer, R. & Moore J.S.[1973], The sharing of structure in theorem proving programs, Machine Intelligence 7, Edin. Univ. Pres.

Bruynooghe,M.[1976], An Interpreter for predicate logic programs, Report CW10, Applied Maths & Prog. Div., Katholieke Univ., Leuven.

Burstall,R.M.[1969], Proving properties of programs by structural induction, Computer Journal, Vol.12, No. 1.

Burstall,R.M. & Darlington,J.[1975], Some transformations for developing recursive programs, Proc. Int. Conf. on Reliable Software, Los Angeles.

Burstall,R.M.[1969], Formal description of program structure and semantics in first order logic, Machine Intelligence 5, Edin. Univ. Press.

Cartwright,R.[1976], User defined types as an aid to verifying LISP programs, 3rd Int. Coll. on Automata, Languages, Programming, Edinburgh.

Clark,K.L.[1977], Verification and Synthesis of logic programs, Research report, CCD, Imperial College.

Clark,K.L. & Darlington,J.[1980], Algorithm classification through synthesis, to appear in the Computer Journal.

Clark,K.L. & McCabe,F.M.[1979], IC-PROLOG reference manual, CCD Research Report, Imperial College, London.

Clark,K.L. & Sickel,S.[1977], Predicate Logic: a calculus for deriving programs, Procs. IJCAI-77, Boston.

Clark,K.L. & Tarnlund,S.[1977], A first order theory of data and programs, Proc. IFIP Congress, Toronto.

Darlington,J.[1975], Application of program transformation to program synthesis, Colloques IRIA on Proving and Improving Programs.

van Emdem,M.H. & Kowalski, R.A.[1976], The semantics of predicate logic as programming language, JACM 23,No.4.

Friedman,D.P. & Wise,D.S.[1976], CONS should not evaluate its arguments, 3rd Int. Conf. on Automata, Languages & Programming, Edinburgh

Green,C.[1969], Application of theorem proving to question answering systems, Ph.D. Thesis, Stanford University.

Hansson, A. & Tarnlund, S-A.[1979], A natural programming calculus, Proc. IJCAI-6, Tokyo.

Hayes,P.J.[1973], Computation and deduction, Procs. MFCS Conf., Czech Academy of Sciences.

Hill,R.[1974], LUSH-resolution and its completeness, Research report, A.I. Dept., Edinburgh University.

Hoare,C.A.R.[1973], Recursive data structures, Research report, Computer Science Dept., Stanford University.

Hogger,C.J.[1977], Deductive synthesis of logic programs, Research report, Theory of Computing research group, CCD, Imperial College.

Hogger,C.J.[1978]m, Derivation of logic programs, Ph.D. Thesis, Imperial College, London.

Kleene,S.[1952], Introduction to Metamathematics, von Nostrand, New York.

Kowalski,R.[1974], Predicate logic as programming language, Proc.IFIP Congress, Stockholm.

Kowalski,R.[1979a], Algorithm = Logic + Control, CACM 22(7).

Kowalski,R.[1979b], Logic for Problem Solving, North Holland, New York.

Manna,Z. & Waldinger,R.J.[1975], Knowledge and reasoning in program synthesis, Artificial Intelligence Journal, 6(2).

Manna,Z. & Waldinger,R.J.[1977], The automatic synthesis of systems of recursive programs, Proc. IJCAI-77, Boston.

McCarthy,J.[1977], Representation of recursive programs in first order logic, Draft paper, A.I. Lab, Stanford University.

Mendelson, E.[1964], An Introduction to Mathematical Logic, Van Nostrand Co. Inc., Princeton.

Morris,J.H. & Henderson,P.[1976], A lazy evaluator, Proceedings of 3rd ACM POPL Conference.

Morris,J.H. & Wegbreit,B.[1977], Subgoal induction, CACM 20, pp 209-222.

Quine, W.V.[1964], Methods of Logic, Routledge & Keegan Paul, London.

Park,D.[1969], Fixpoint induction and proofs of program properties, Machine Intelligence 5, Edinburgh Univ. Press.

Patterson, M.S. & Wegman,M.N.[1976], Linear unification, Proc. 8th Ann. ACM Symp. on Theory of Computing.

Roberts,G.[1977], An Implementation of PROLOG, M.Sc. Thesis, University of Waterloo.

Robinson,J.A.[1965], A machine orientated logic based on the resolution principle, JACM 12, 1.

Robinson,J.A.,[1979], Logic: form and function, Edinburgh University Press, Edinburgh.

Roussel,,P.[1975], PROLOG: Manuel de reference et d'utilisation, Groupe d'Intelligence Artificielle, UER de Luminy, Marseilles.

Schwarz,J.[1977], Using annotations to make recursion equations behave, Research report, A.I. Dept., Edinburgh University.

Scott,D.[1970], Outline of a mathematical theory of computation, Technical Monograph PRG-2, Oxford University Computing Lab., Programming Research Group.

Stoy,J.E.[1977], Denotational Semantics, MIT Press, Cambridge, Mass.

Suppes, P.[1957], Introduction to Logic, Van Nostrand Reinhold, New York.

Sussman,G.J., & Winograd,T.[1970], MICRO-PLANNER Reference Manual, AI Memo 203, MIT AI Lab.

Szeredi, P.[1977], PROLOG-a very high level language based on predicate logic, 2nd Hung. conf. on comp. Science, Budapest.

Tarski,A.[1955], A lattice theoretic fixpoint theorem and its applicatiobns, Pacific Journal of Maths,5 pp 285-309.

Warren,D.[1977], Implementing PROLOG - compiling predicate logic programs, Research reports 39,40, AI Dept., Edinburgh University.

Warren, D.[1979], PROLOG on the DECsystem-10, Proc. AISB Summer School on Expert Systems, to be published by Edinburgh Univ. Press.

Warren,D., Pereira,L., Pereira,F.[1977], PROLOG: the language and its implementation compared to LISP, SIGPLAN/SIGART Prog. Lang. Conf., Rochester.