Today

School of informatics — The University of Edinburgh

- Prolog interpreter algorithms
- Beyond Pure Prolog: “meta”-predicates
- Closed World Assumption & Negation as Failure.

Algorithms for definite clause interpreter

We have seen the outline of how inference in definite clause logic can be automated. Let's spell out a bit more concretely some of the key procedures involved.

These will be given by Haskell functions, with comments. Haskell is a functional programming language – see overview material[1].

An implementation of a basic Prolog interpreter in Haskell is also available[2].

Features in common with other languages, such as parsing, pretty printing, input/output must be dealt with, but we concentrate on the key steps in inference and search.

Acknowledgements to Mark Jones for the Haskell code.

[1] `http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/#info`

[2] `http://darcs.haskell.org/nofib/real/prolog`

For an interpreter, there is no need to make a distinction between function symbols and predicates. Here are the basic data-types:

```
type Id       = (Int,String)
          -- variable identifiers, Int allows renaming
type Atom     = String
          -- for constant, fn symbol or predicate
data Term     = Var Id | Struct Atom [Term]
          -- Var, Struct constructors for
          --              pattern matching
data Clause   = Term :== [Term]
          --  Clause is written as "tm :== [tm,tm,...]"
data Database = Db [(Atom,[Clause])]
          --  The program
```

Since haskell is a functional language, in which functions are first-class objects, substitutions can be treated directly as functions from (some) variables to terms.

```haskell
--- Substitutions:

type Subst = Id -> Term

-- subsns taken as fns mapping variable ids to terms.
--
-- apply s   extends subsn s to take terms to terms
-- nullSubst is identity subsn
--
-- i ->> t   maps the variable id i to the term t,
--           but otherwise behaves like nullSubst.
-- s1 @@ s2  is the composition of subsns s1 and s2
```

Substitution Operations

```
apply                         :: Subst -> Term -> Term
apply s (Var i)               = s i
apply s (Struct a ts)         = Struct a (map (apply s) ts)
  -- apply the substitution recursively to every arg

nullSubst                     :: Subst
nullSubst i                   = Var i

(->>)                         :: Id -> Term -> Subst
(->>) i t j | j==i            = t           -- case j==i
            | otherwise       = Var j       -- any other case

(@@)                          :: Subst -> Subst -> Subst
s1 @@ s2                      = (apply s1) . s2
            -- "." is function composition;
            --  (f . g) x = f(g(x))
```

Unification with occurs check

success is a singleton list with mgu, failure is empty list.

```
unify :: Term -> Term -> [Subst]
    -- unify takes two terms, returns list of subsns

unify (Var x) (Var y)
      = if x==y then [nullSubst] else [x->>Var y]
unify (Var x)  t2
      = [ x ->> t2 | not (x `elem` varsIn t2) ]
    -- [] if x is in t2, otherwise [ x ->> t2]
unify t1   (Var y)
      = [ y ->> t1 | not (y `elem` varsIn t1) ]
unify (Struct a ts) (Struct b ss)
      = [ u | a==b, u<-listUnify ts ss ]
    -- [] if a =/=b, otherwise call listUnify on args
```

Unification ctd

```
listUnify :: [Term] -> [Term] -> [Subst]

listUnify []     []     = [nullSubst]
listUnify []     (r:rs) = []
    -- fail if lists of different length
listUnify (t:ts) []     = []
listUnify (t:ts) (r:rs) =
     [ u2 @@ u1 | -- compose subs u1, u2, where
        u1<-unify t r, -- u1 is unifier of t,r
        u2<-listUnify (map (apply u1) ts)
                      (map (apply u1) rs) ]
    -- apply u1 to all remaining arguments,
    -- and call recursively to get u2
```

The Proof Search Space

```
data Prooftree = Done Subst  |  Choice [Prooftree]
-- Done [] is failure,
-- Done [s] suceeds with subsitution s,
-- Choice is a list of open possible derivations
-- prooftree gives proof search tree for a given goal;
-- since Haskell is lazy, doesn't expand trees here.
prooftree   :: Database -> Int -> Subst -> [Term]
                                    -> Prooftree
```

Proof Search Space (2)

```
prooftree db = pt
 where pt    :: Int -> Subst -> [Term] -> Prooftree
             -- proof depth, result so far, list of
                goals
   pt n s [] = Done s
   pt n s (g:gs) = Choice
      [ pt (n+1) (u@@s) (map (apply u) (tp++gs))
      | (tm:==tp)<-renClauses db n g, u<-unify g tm ]
  --  for each clause with head unifiable with
  --  1st goal, get new goal list: add clause body
  --  at FRONT of goals (to get depth first), and
  --  apply unifier;  also update accumulated subsn
```

Proof Search

```
-- do depth-first search of a proof tree,
-- producing the list of solution substitutions
-- as they are encountered.
search                  :: Prooftree -> [Subst]
search (Done s)         = [s]  -- found a solution!
search (Choice pts) = [ s | pt <- pts, s <- search pt ]
            -- look successively at each tree in pts,
            -- call search recursively on it

prove     :: Database -> [Term] -> [Subst]
            -- initialise the search
prove db  = search . prooftree db 1 nullSubst
```

Meta-language

When we use one language to talk about another language, we say that the meta-language is used to talk about the object language.

Examples

English as meta-language, with French as object language:

> *The word "poisson" is a masculine noun.*

English as meta-language, with English as object-language:

> *It is hard to understand "Everything I say is false".*

Examples ctd

Prolog contains a mixture of object-level and meta-level statements.

`father(a,b).`	object-level
`functor(father(a,b),father,2).`	meta-level
`var(X).`	meta-level

It is better to keep these uses conceptually distinct.
We have seen that `var/1` does not function according to Prolog's declarative semantics.

Take the program:

```
father(a,b).

ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- father(X,Z), ancestor(Z,Y).
```

We can write a description of Prolog programs in Prolog:

```
clause( father(a,b), true ).
clause( ancestor(X,Y), father(X,Y) ).
clause( ancestor(X,Y),
        (father(X,Z), ancestor(Z,Y)) ).
```

This treatment of Prolog in Prolog also breaks the declarative reading.

The statement `clause( father(a,b), true )` cannot be parsed in definite clause logic so that `father` is a predicate – it can only be a function symbol.

One possibility is to consider that we are dealing with two languages – an object language in which `father` is a predicate, and a meta-language which talks about the object language, and where `clause` is a predicate.

This make it hard to understand in a declarative way programs where the two languages are mixed. The language Goedel[3] developed a systematic approach to logic programming with two interconnected languages.

[3] `https://en.wikipedia.org/wiki/Gödel_(programming_language)`

Negation by failure

Prolog does not distinguish between being unable to find a derivation, and claiming that the query is false; that is, it does not distinguish between the “false” and the “unknown” values we have above.

When we take a Prolog response of `no.` as indicating that a query is false, we are making use of the idea of negation as failure: if a statement cannot be derived, then it is false.

Clearly, this assumption is not always valid! If some information is not present in the program, failure to find a derivation should not let us conclude that the query is false – we just don’t have the information to decide.

Knowing the answers

A good situation to be in is where we have enough information to answer any possible query. If we know

$$\begin{array}{rcl} & & poor(jane) \\ poor(jane) & \rightarrow & happy(jane) \\ & & happy(fred) \end{array}$$

we do not know enough to answer the query

$$?- \ poor(fred)$$

Complete Theories

We say a theory T is complete (for ground atoms) if and only if:

> *for every (ground atom) query (eg $poor(fred)$),*
> *we can prove either $poor(fred)$ or $\neg poor(fred)$.*

A ground atom is a statement of the form $P(t_1, \ldots, t_n)$ where there are no variables in any t_i; so it is a basic statement about particular objects.

NB, this is yet another different use of the term complete (compare complete inference system, complete search strategy).

Complete theories (2)

Our example T is not complete in this sense; we can extend it to make a complete T using the Closed World Assumption (CWA).

The idea is to add in the *negation* of a ground atom whenever the ground atom cannot be deduced from the KB.
This makes the assumption that

> *all the basic positive information about the domain follows from what is already in T.*

Here basic positive information refers to atomic ground statements.

CWA as an augmented T

We can define the effect of the CWA using the standard logic we saw earlier. Given a T written in first-order logic, we augment T to get a bigger set of formulas $CWA(T)$; the extra formulas we add are:

$$X_T = \{ \neg p(t_1, \ldots, t_n) : t_1, \ldots, t_n \; ground, \textbf{not} \; T \vdash p(t_1, \ldots, t_n) \}$$

Now we can define what it is to follow from T using CWA:
a formula Q follows from T using the CWA iff

$$T \cup X_T \models Q$$

Example

In the example, we can now conclude $\neg poor(fred)$, since from the original T we *cannot* show $poor(fred)$. Thus we have $\neg poor(fred)$ is in X_T.

In fact, in this case

$$X_T = \{ \neg poor(fred) \},$$

assuming there are no other constants in the language except $jane, fred$. In this case, we can compute the set X_T by looking at all possibilities. In general though the set X_T may be infinite, so this is not a computable way to realise the CWA.

One use of CWA is in looking at a failed Prolog query of the form

```
?- property(t1,t2).
```

as saying that the query is in fact false.

CWA gives completeness for ground atoms

For any definite clause theory, the extended theory:

$$CWA(T) = T \cup \{ \neg p(t_1, \ldots, t_n) : t_1, \ldots, t_n \textit{ ground}, \\ \textbf{not } T \vdash p(t_1, \ldots, t_n) \}$$

is complete for ground atoms.

This is simply because for such a query Q, if Q is not a logical consequence of T, then $\neg Q$ is in the extended $CWA(T)$, and so $\neg Q$ is a consequence of $CWA(Q)$.

Summary

- Prolog interpreter algorithms
- Beyond Pure Prolog: “meta”-predicates
- Closed World Assumption