# Today

- 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

# Representing statements

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 are constructors for pattern matching
data Clause   = Term :== [Term]
                --  Clause is written as " tm :== [tm,tm,...] "
data Database = Db [(Atom,[Clause])]
                --  The program
```

# Substitutions

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.

```
--- Substitutions:

type Subst = Id -> Term

-- substitutions are represented by functions mapping variable ids to terms.
--
-- apply s   extends the substitution s to a function mapping terms to terms
-- nullSubst is the empty substitution which maps every identifier to the
--           same identifier (as a term).
-- i ->> t   is the substitution which maps the variable id i to the term t,
--           but otherwise behaves like nullSubst.
-- s1 @@ s2  is the composition of substitutions 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 argmnt

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, and returns a list of substitutions

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
```

```
data Prooftree = Done Subst  |  Choice [Prooftree]
     -- Done [] is failure, Done [s] suceeds with subsitution s,
     -- Choice is a list of open possible derivations

-- prooftree constructs a suitable proof search tree for a specified goal
-- since Haskell is lazy, doesn't expand trees here!
prooftree   :: Database -> Int -> Subst -> [Term] -> Prooftree
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 against first goal,
                    --  get new goal list: add clause body at FRONT of goals
                    --  (to get depth first), and apply unifier;  also
                    --  update accumulated substitution
```

# Proof Search

```
-- search performs a 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
```

This is the basic engine to find the *first* solution to a query. An interpreter that deals with subsequent solutions, and with cuts, is not much more complicated; see `Engine.hs` for the extended interpreter.

# Meta-language

Thus we get two languages, one describing the other. 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 distinct.

Notice that `var/1` does not function according to Prolog's declarative semantics:

Compare:

```
| ?- var(X),X=2.

X = 2 ?

yes
| ?- X=2, var(X).

no
```

Remember, Prolog comma is just conjunction –
the two queries are logically equivalent, so the answers should be the same.

So this behaviour is inconsistent with the declarative reading.

# Prolog in Prolog

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)) ).
```

# Status of meta-predicates

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] `http://www.scs.leeds.ac.uk/hill/GOEDEL/expgoedel.html`

# 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) iff for every query (like $poor(fred)$) we can conclude 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.

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* $T$.

# 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:

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

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.

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.

# Summary

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