Tableau method for FOL

Logical rules for quantifiers

- We saw in the sequent calculus that logical rules for quantifiers depend upon instantiating the quantified variable with terms.

- The rules allow us to use new variables provided they don’t appear elsewhere in the proof
  - Sometimes elsewhere is implicit

Quantifiers

- The bound variables in quantifiers are meant to range over the complete domain D
- The resulting terms for forall are meant to be “and”ed together
- The resulting terms for exists are meant to be “or”ed together
- In a symbolic system, if we treat a single variable with the right rules we can get both kinds of effects.

Forall

- Consider the term $(\forall\ x\ .\ F)$
- Lets invent a new variable "y" which is fresh to the program.
- If we can prove F>>=(x|->y) without any assumptions about y, then we have proven it for $(\forall\ x\ .\ F)$

Exists

- Existentials ($\exists$ x . F) are more subtle.
- In an existential we don't need to prove it for all occurrences of x, but for some unknown x which makes F true, but x must be in the domain.
- Skolem functions provide the solutions

Skolem functions

- Consider the formula
- $(\exists\ x\ .\ Odd(x)\ /\backslash\ x=y+1\ /\backslash\ Even(y))$
- We don’t know what the x is, but it probably depends upon y, so lets invent a function F such that x=F(y). Then we have
- $Odd(F(y))\ /\backslash\ F(y)=y+1\ /\backslash\ Even(y))$
- Note that the variable x has dissappeared!

Rules for skolem functions

- Consider (∃ x . F)
- Let the free variables of F be (a,b,c,x)
- Then we may invent a skolem function, g, whose arguments are all the variables, exceopt for x, which is (a,b,c).
- So we get F>>=(x |-> g(a,b,c)) provided g is a new function symbol.
- What can we assume about g(a,b,c), nothing except that it is equal to g(a,b,c)!

Parameters

- Let L(c,f,p) be a logic.
- Invent a new set of constant symbols (disjoint from c and f) called parameters
- Let L^{par} be the logic L(c U *par*, f, p)
- Let C be a collection of sentences (closed formula) of Let L^{par}
- We'd like C to have some properties along the lines of the Hintika sets of propositional logic

Herbrand Models

- Parameters are constants.
- Since they are not part of any original model of L, we don't know how to model L^{par}
- Luckily we can invent a model, called a Herbrand model, which has all the properties we need a model to have
- The Herbrand model is often called a string (or term) model.

- In a Herbrand model substitutions and assignments coincide
- A substitution maps variables to terms
- An assignment maps variables to the Model set D
- In a Herbrand model D is the set of terms.
- We won’t do it here, but we can prove that the Herbrand model has an interesting property called first order consisteny.

First order consistency

$\mathcal{C}$ *is a* first-order consistency property *if, for each* $S \in \mathcal{C}$:

1. *For every atomic proposition* ϕ *at most one of* ϕ *or* $\neg\phi$ *is in* S.
2. $\bot \notin S$, $\neg\top \notin S$.
3. *If* $\neg\neg\phi \in S$ *then* $S \cup \{\phi\} \in \mathcal{C}$.
4. *If* $\alpha \in S$ *then* $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.
5. *If* $\beta \in S$ *then* $S \cup \{\beta_1\} \in \mathcal{C}$ *or* $S \cup \{\beta_2\} \in \mathcal{C}$.
6. *If* $\gamma \in S$ *then* $S \cup \{\gamma(t)\} \in \mathcal{C}$ *for every closed term* t *of* L^{par}.
7. *If* $\delta \in S$ *then* $S \cup \{\delta(p)\} \in \mathcal{C}$ *for some parameter* p *of* L^{par}.

Discriminating formulas in FO logic

- As in predicate logic we can discriminate formulas into certain sets (e.g Alpha, Beta, Lit)

- We need two new categories
  - Gamma for $(\forall x . F)$ and $\sim(\exists x . F)$
  - Delta for $(\exists x . F)$ and $\sim(\forall x . F)$

What does this look like over Formula?

- In the propositional calculus we needed to deal only with variables and the connectives.
- In the predicate calculus we have predicates, connectives, and the quantifiers.

```
data Discrim v a
    = Alpha a a
    | Beta a a
    | Lit a
    | Gamma v a
    | Delta v a
 deriving Show

notP (Conn Not [x]) = x
notP x = Conn Not [x]
```

```
discrim :: Formula p f v -> Discrim v (Formula p f v)
discrim (p@(Rel r ts)) = Lit p
discrim (Conn T []) = Lit (Conn T [])
discrim (Conn F []) = Lit (Conn F [])
discrim (Conn And [x,y]) = Alpha x y
discrim (Conn Or [x,y]) = Beta x y
discrim (Conn Imp [x,y]) = Beta (notP x) y
discrim (Conn Not [x]) =
  case x of
    (Rel r ts) -> Lit(notP x)
    (Conn T []) -> Lit(Conn F [])
    (Conn F []) -> Lit(Conn T [])
    (Conn And [x,y]) -> Beta (notP x) (notP y)
    (Conn Or [x,y]) -> Alpha (notP x) (notP y)
    (Conn Imp [x,y]) -> Alpha x (notP y)
    (Conn Not [x]) -> discrim x
    (Quant All v f) -> Delta v (notP f)
    (Quant Exist v f) -> Gamma v (notP f)
discrim (Quant All v f) = Gamma v f
discrim (Quant Exist v f) = Delta v f
```

Tableau Method

- The tableau method exploits this by building a branching tree, such that every path maintains this property.
- Differences from tableau for propostional logic.
  1. We no longer have propositional variables, but now have predicates over terms.
  2. We have to deal with quantifiers
  3. To close a path, we need conflicting predicates.

Properties of discrimination

- Discrimination splits all sentences into one of 5 sets.
- All sentences in each set have the same properties
- The splitting is arranged so each set has exactly one of the first order consistency properties

Tableau Trees

```
data Tree
  = Direct (FormulaS) Tree
  | Branch Tree Tree
  | Leaf
  | Closed FormulaS FormulaS
```

A Leaf marks the end of a path that can be extended

A (Closed x y) marks the end of a closed path where x and y are conjugates (appearing n the path ended by (Closed x y)

Extending a tree

```
extendTree p Leaf = p
extendTree p (Direct q t) =
   Direct q (extendTree p t)
extendTree p (Branch x y) =
   Branch (extendTree p x) (extendTree p y)
extendTree p (Closed x y) =
   Closed x y

-- make 1 and 2 elements trees
single p = Direct (p) Leaf
double p q = Direct (p) (single q)
```

The algorithm

- We start with the negation of the formula to be proved.
- We have a list of pending nodes (not yet visited) and a current tree.
- Pick an unvisited node, discriminate on it, and extend the tree according to the rules
- We need to do this in a state monad as we must be able to invent new variables and skolem functions not in the term.

```haskell
tabTree :: [FormulaS] -> Tree -> State Int Tree

tabTree [] tree = return tree
tabTree (x:xs) tree =
  case discrim x of
    Lit p -> tabTree xs tree
    Alpha a b -> tabTree (a:b:xs) (extendTree (double a b) tree)
    Beta a b ->
      do { x <- tabTree (a:xs) (single a)
         ; y <- tabTree (b:xs) (single b)
         ; return(extendTree (Branch x y) tree) }
    Gamma s f ->
      do { t <- freshTerm
         ; let form = (subst (s |-> t) f)
         ; tabTree (form:xs) (extendTree (single form) tree) }
    Delta s f ->
      do { t <- freshSkolem s f
         ; tabTree (t:xs) (extendTree (single t) tree)}
```

Example

- ((ALL x. O(x, x)) --> (ALL x. (EX y. O(x, y))))

```
~((ALL x. O(x, x)) --> (ALL x. (EX y. O(x, y))))
    (ALL x. O(x, x))
~(ALL x. (EX y. O(x, y)))
     O(n1, n1)
~(EX y. O(f2(), y))
~O(f2(), n3)
```

How do we know if this tree can be closed?

```
~((ALL x. O(x, x)) --> (ALL x.
  (EX y. O(x, y))))
     (ALL x. O(x, x))
~(ALL x. (EX y. O(x, y)))
      O(n1, n1)
~(EX y. O(f2(), y))
~O( f2(), n3)
```

```
~((ALL x. O(x, x)) --> (ALL x. (EX y. O(x, y))))
      (ALL x. O(x, x))
   ~(ALL x. (EX y. O(x, y)))
        O(f2(), f2())
      ~(EX y. O(f2(), y))
        ~O(f2(), f2())
X(~O(f2(), f2()),O(f2(), f2()))
```

Unification

- unification tries so see if two terms can be made identical by applying the same substitution.

- It works by finding two terms that differ only by a variable in one and a term in the other.

unify

```
unify (Var v) (Var u)
  | u==v = return emptySubst
unify (Var v) y =
  do { occurs v y
     ; return(v |-> y)}
unify y (Var v) =
  do { occurs v y
     ; return (v |-> y) }
unify (Fun _ f ts) (Fun _ g ss)
  | f==g = unifyLists ts ss
unify x y = Nothing
```

UnifyLists

```
unifyLists [] [] = Just emptySubst
unifyLists [] (x:xs) = Nothing
unifyLists (x:xs) [] = Nothing
unifyLists (x:xs) (y:ys) =
  do { s1 <- unify x y
     ; s2 <- unifyLists
               (map (subTerm s1) xs)
               (map (subTerm s1) ys)
     ; return(s2 |=> s1)}
```

```
unifyForm (Rel x ts) (Rel y ss)
  | x==y = unifyLists ts ss
unifyForm (Conn c1 ts) (Conn c2 ss)
  | c1==c2 = unifyForms ts ss
unifyForm x y = Nothing

unifyForms [] [] = Just emptySubst
unifyForms [] (x:xs) = Nothing
unifyForms (x:xs) [] = Nothing
unifyForms (x:xs) (y:ys) =
  do { s1 <- unifyForm x y
     ; s2 <- unifyForms (map (subst s1) xs)
                        (map (subst s1) ys)
     ; return(s2 |=> s1)}
```

Subtle

- Consider
- ((ALL x. O(x, x)) --> (ALL x. (ALL y. (O(x, x) | O(y, y)))))

- What is the tableau?
- Does it close

Problem

- What about this example

((ALL x. O(x, x)) --> (ALL x. (ALL y. (O(x, x) & O(y, y)))))

The tableau

```
~((ALL x. O(x, x)) --> (ALL x. (ALL y. (O(x, x) & O(y, y)))))
          (ALL x. O(x, x))
~(ALL x. (ALL y. (O(x, x) & O(y, y))))
            O(n1, n1)
~(ALL y. (O(f2(), f2()) & O(y, y)))
 ~(O(f2(), f2()) & O(f3(), f3()))
+-------------+  +-------------+
|~O(f2(), f2())|  |~O(f3(), f3())|
+-------------+  +-------------+
```

- The problem with the last example is that a forall term is instantiated at one variable
- But as we saw in the sequent calculus we can instantiate it several times.
- But if we’re not carefull we may go into a infinite loop. Why?

Paths

- We don’t need to actually compute the tree
- Only the paths of literal terms are necessary

- As we saw I the propositional case, the order we visit the nodes also matters.

```
tab3:: [FormulaS] -> [[FormulaS]] -> State Int
[[FormulaS]]
tab3 [] paths = return paths
tab3 (x:xs) paths =
  case discrim x of
    Lit p -> tab3 xs (map (cons3 p) paths)
    Alpha a b -> tab3 (insert3 a (insert3 b xs))
                      (map (cons3 a . cons3 b)
paths)
    Beta a b ->
        do { ms <- tab3 (insert3 a xs)
                        (map (cons3 a) paths)
           ; ns <- tab3 (insert3 b xs)
                        (map (cons3 b) paths)
           ; return (ms++ns)}
    Gamma v f ->
       do { t <- freshTerm
          ; let form = (subst (v |-> t) f)
          ; tab3 (form:xs) paths }
    Delta s f ->
      do { t <- freshSkolem s f
         ; tab3 (t:xs) paths }
```