Tutorial on Theorem Proving

Jan van Eijck

Abstract

This tutorial gives an overview of basic concepts of first order theorem proving implemented in a functional language (Haskell): substitution, unification, skolemization, resolution, tableau construction. The paper discusses the role of substitution and unification in reasoning processes involving variables. These ingredients play an important role in the reasoning engines behind functional programming languages such as Haskell and logic programming languages such as Prolog. They are also crucial for the design of engines for automated theorem proving. At the end of this tutorial we will implement a basic Prolog engine based on the resolution rule and a tableau-style theorem prover.[1]

```
module TOTP

where

import Data.List
```

1 Representing Substitutions

Substitution is the replacement of expressions in expressions by other expressions. To see that substitutions are a fundamental (perhaps *the* fundamental) issue in computation, consider a programming language with basic programming instructions of the form $v \mapsto e$, where v is a variable, and e is an expression that may itself contain variables. Then the command $P \mapsto Q$

[1] The Haskell code in this chapter is based in part on a Prolog implementation in Haskell written by Mark P. Jones that (once) was distributed as part of the *hugs* demo programs. Studying this implementation is highly recommended.

would allow us to get $Q \Rightarrow (Q \Rightarrow Q)$ from $P \Rightarrow (Q \Rightarrow P)$. We denote the process of applying the substitution $v \mapsto E$ to an expression E' as $\{v \mapsto E\}E'$.

The command $n \mapsto (Sn)$ applied to (Sn) gets us $(S(Sn))$, the same command applied to $\mathit{plus}((S(Sm)n$ gets us $\mathit{plus}((S(Sm)(Sn)$, and so on.

Function application is itself a substitution process. Successor is the function $\lambda n.Sn$. Applying this function to an argument a is a matter of substituting a for n in Sn. We can view the application process as a substitution $[n \mapsto a](Sn)$. Evaluation of functional expressions is a matter of performing substitutions.

A first thing to note is that the order in which we perform substitutions may make a great difference. Consider again $P \Rightarrow (Q \Rightarrow P)$ as a starting point. Suppose we first perform $P \mapsto (Q \Rightarrow Q)$, and next $Q \mapsto P$. This yields:

$$(P \Rightarrow P) \Rightarrow (P \Rightarrow (P \Rightarrow P)).$$

Suppose on the other hand that, starting out from the same formula, we first perform $Q \mapsto P$ and next $P \mapsto (Q \Rightarrow Q)$. This yields:

$$(Q \Rightarrow Q) \Rightarrow ((Q \Rightarrow Q) \Rightarrow (Q \Rightarrow Q)).$$

Finally, we may consider performing the substitutions *simultaneously*. The result is:

$$(Q \Rightarrow Q) \Rightarrow (P \Rightarrow (Q \Rightarrow Q)).$$

Different notations are used for the application of a substitution to an expression. We will write the substitution as a function preceding its argument (this is useful because it agrees with the implementation in functional programming that we will consider below).

We will employ $\{v_1 \mapsto E_1, \dots, v_n \mapsto E_n\}E$ for the simultaneous substitution of E_1 for v_1, $\dots$, E_n for v_n in E. Thus, we have:

$$\{P \mapsto Q, Q \mapsto (Q \Rightarrow Q)\}(P \Rightarrow (Q \Rightarrow P) = (Q \Rightarrow ((Q \Rightarrow Q) \Rightarrow Q)).$$

We have already seen that in general substitutions do not commute: the result of performing a first substitution, and next a second one is not always the same as first performing the second one and next the first one. For another example: $\{y \mapsto z\}$ after $\{x \mapsto y\}$ is equivalent to the simultaneous substitution $\{x \mapsto z, y \mapsto z\}$, $\{x \mapsto y\}$ after $\{y \mapsto z\}$ is equivalent to the simultaneous substitution $\{x \mapsto y, y \mapsto z\}$.

The example also explains why it is useful to allow simultaneous substitution next to single variable substitution. This is because we sometimes want to perform one substitution after another, and the result of combining two single variable substitutions will in general not be a single variable substitution. For example, first performing $\{x \mapsto y\}$ and next $\{y \mapsto z\}$ has the same effect as performing the simultaneous substitution $\{x \mapsto z, y \mapsto z\}$.

Representing Identifiers, Terms, Formulas We will study substitution in the context of formal languages, so first we have to make up our minds about how to represent the ingredients of these languages. In this section we define predicate logical formulas as a Haskell data type.

Suppose a set of variables V and a set of function names F are given. Then the recursive definition of the language of terms over these is:

- If v is a variable in V, v is a term.
- If f is a function name in F and $[t_1, \dots, t_n]$ is a list of terms then $f[t_1, \dots, t_n]$ is a term.

This definition is to be understood with an implicit closure condition: nothing else is a term.

A function f with two arguments t_1 and t_2 gets represented as $f[t_1, t_2]$. We can represent almost anything that we have encountered so far in this book as a term. Think of a constant c as a function of the form $c[]$. Also, we have seen that we can look at the propositional connectives as function symbols, so $P \wedge Q$ can get represented as $(\wedge)[P, Q]$. A Haskell function *fct* of type $a \to b \to c$ can be represented with its two arguments x and y as $fct[x, y]$. A binary relation symbol R with two arguments x and y gets represented as $R[x, y]$. A conjunction $Rxy \wedge Ryz$ gets represented as $(\wedge)[R[x, y], R[y, z]]$, and so on.

To implement this, we start out from *identifiers* (type `Id`), pairs of strings and indices. The indices that we use in identifiers are needed later on for tree indexing, so we implement them as lists of integers.

```
type Name     = String
type Index    = [Int]
data Id       = Id Name Index deriving (Eq,Ord)
```

It is useful to put `Id` in the `Show` class:

```
instance Show Id where
  show (Id name [])  = name
  show (Id name [i]) = name ++ ('_': show i)
  show (Id name is)  = name ++ ('_': showInts is)
     where showInts [] = ""
           showInts (i:is) = show i ++ showInts is
```

Some examples of variable indices:

```
ix = Id "x" []
iy = Id "y" []
iz = Id "z" []
```

Terms are now given by:

```
data Term     = Var Id | Struct Id [Term] deriving Eq
```

It is useful to have an ordering on terms:

```
instance Ord Term where
  compare (Var x) (Var y) = compare x y
  compare (Var x) _       = LT
  compare _       (Var y) = GT
  compare (Struct a ts) (Struct b rs) | a == b = compare ts rs
                                      | a < b  = LT
                                      | a > b  = GT
```

Some examples of variable terms:

```
x    = Var ix
y    = Var iy
z    = Var iz
```

Some examples of constant terms:

```
a     = Struct (Id "a" []) []
b     = Struct (Id "b" []) []
c     = Struct (Id "c" []) []
zero  = Struct (Id "z" []) []
s     = Struct (Id "s" [])
t     = Struct (Id "t" [])
u     = Struct (Id "u" [])
one   = s[zero]
two   = s[one]
three = s[two]
four  = s[three]
five  = s[four]
```

Here is the declaration of `Term` as an instance of the `Show` class:

```
instance Show Term where
  show (Var id)         = show id
  show (Struct id [])   = show id
  show (Struct id ts)   = show id ++ concat [ show ts ]
```

The function `isVar` checks whether a term is a variable:

```
isVar :: Term -> Bool
isVar (Var _) = True
isVar _       = False
```

The functions `varsInTerm` and `varsInTerms` give the variables that occur in a term or a term list. Variable lists should not contain duplicates; the function `nub` cleans up the variable lists.

```
varsInTerm :: Term -> [Id]
varsInTerm (Var i)       = [i]
varsInTerm (Struct i ts) = varsInTerms ts

varsInTerms :: [Term] -> [Id]
varsInTerms = nub . concat . map varsInTerm
```

Once we have identifiers and terms, it is straightforward to declare a data type for the language of predicate logic. We assume that **P** is a set of predicate names. For convenience we will represent conjunction and disjunction as operations on lists of formulas. We use prefix notation for implications and equivalences.

- If P is a predicate name in **P**, and $[t_1, \ldots, t_n]$ is a list of terms then $P[t_1, \ldots, t_n]$ is an atom.
- If A is an atom then A is a formula.
- If F is a formula then $\neg F$ is a formula.
- If F_1 and F_2 are formulas,
  then then $(\Rightarrow)F_1F_2$, $(\Leftrightarrow)F_1F_2$ are formulas.
- If $[F_1, \ldots, F_n]$ is a list of formulas,
  then $(\wedge)[F_1, \ldots, F_n]$ and $(\vee)[F_1, \ldots, F_n]$ are formulas.
- If F is a formula and v is a variable then $\forall vF$ and $\exists vF$ are formulas.

Again we have an implicit closure condition: nothing else is a formula.

Exercise 1 *What are the truth conditions for generalized conjunctions and disjunctions? Applying these to the special cases* `Conj []` *and* `Disj []`*, what is the result?*

Exercise 2 *Give the free occurrences of x in the formula*

$$(\wedge)[P[x], \exists x(\vee)[R[x,y], S[x,y,z]]].$$

A data type for quantified formulas can be declared in Haskell as follows (for convenience, we use identifiers as predicate names):

```
data Form =  Atom Id [Term]
           | Eq Term Term
           | Neg Form
           | Impl Form Form
           | Equi Form Form
           | Conj [Form]
           | Disj [Form]
           | Forall Id Form
           | Exists Id Form
     deriving (Eq,Ord)
```

Note that an ordering on formulas is derived from the ordering on terms.

A `show` function for formulas:

```
instance Show Form where
  show (Atom id []) = show id
  show (Atom id ts) = show id ++ concat [ show ts ]
  show (Eq t1 t2)   = show t1 ++ "==" ++ show t2
  show (Neg form)   = '-': (show form)
  show (Impl f1 f2) = "(" ++ show f1 ++ "==>" ++ show f2 ++ ")"
  show (Equi f1 f2) = "(" ++ show f1 ++ "<=>" ++ show f2 ++ ")"
  show (Conj fs)     = "*(" ++ showLst fs ++ ")"
  show (Disj fs)     = "+(" ++ showLst fs ++ ")"
  show (Forall id f) = "A" ++  show id ++ (' ' : show f)
  show (Exists id f) = "E" ++  show id ++ (' ' : show f)

showLst,showRest :: [Form] -> String
showLst [] = ""
showLst (f:fs) = show f ++ showRest fs
showRest [] = ""
showRest (f:fs) = ',': show f ++ showRest fs
```

Exercise 3 *Give a implementation of a function* `freeVarsInForm :: Form -> [Id]` *that gives the list of variables with free occurrences in a formula.*

Representing Substitutions Assume we have a set of variables V, and a set of expressions E with $v \subseteq E$. Then a substitution σ is a finite set of bindings, where a binding is a pair $v \mapsto e$, with $v \in V$ and $e \in E$. Alternatively, a substitution can be viewed as a function $\sigma : V \to E$ with the property that only finitely many members of V are affected. In other words, $\sigma : V \to E$ is a substitution if $\{v \mid \sigma(v) \neq v\}$ is a finite set. For $\sigma : V \to E$ we define $\mathrm{Dom}\,(\sigma) := \{v \mid \theta(v) \neq v\}$, and $\mathrm{Rng}\,(\sigma) := \{\theta(v) \mid \theta(v) \neq v\}$.

For extending a substitution $\sigma : V \to E$ to a mapping $\sigma^\circ : E \to E$, i.e., a map from expressions to expressions, we have to know a bit more about the nature of the expressions. If the expressions contain binders, we have to make sure that we only replace free occurrences of the variables. To keep matters simple, let us at first assume that we have a language without binders, e.g., a language of terms as defined in the previous section.

Here is a data type declaration for substitutions as lists of bindings.

```
type Subst = [(Id,Term)]
```

The identity substitution ϵ (the substitution that changes nothing) gets defined as:

```
epsilon :: Subst
epsilon = []
```

Domain and range of a substitution.

```
dom :: Subst -> [Id]
dom = map fst

rng :: Subst -> [Term]
rng = map snd
```

The restriction of a substitution σ (viewed as a set of binders) to a set of variables W is the substitution

$$\sigma - \{(v, \sigma v) \mid v \notin W\}.$$

Here is the implementation:

```
restriction :: [Id] -> Subst -> Subst
restriction ids = filter (\ (x,_) -> elem x ids)
```

Application of a substitution to an identifier (in fact, a conversion from the representation as a list of binders to the representation as a map from variables to terms):

```
appI :: Subst -> Id -> Term
appI []          y             = (Var y)
appI ((x,x'):xs) y | x == y    = x'
                   | otherwise = appI xs y
```

Exercise 4 *Is it also possible to convert in the other direction? What difficulty do you encounter?*

Lifting of objects σ of type `Subst` to functions of type `Term -> Term` is done by means of:

- $\sigma^\circ(v) := \sigma(v)$.
- $\sigma^\circ(f[t_1, \ldots, t_n]) := f[\sigma^\circ(t_1), \ldots, \sigma^\circ(t_n)]$.

We will often write σ° as σ. Thus, if t is a term and σ a substitution then we write σt for the term obtained from t by simultaneously replacing every occurrence of a variable $v \in \text{Var}\,(t)$ by its θ image θv. To give an example:

$$\{x \mapsto g[x,y], y \mapsto z\}\ g[x,y] = g[g[x,y],z].$$

Exercise 5 *Which terms are meant by the following:*

1. $\{x_1 \mapsto f[x_1]\}\ g[x_1, f[x_1]]$,
2. $\{x_1 \mapsto y_1\}\ g[y_1, f[x_1]]$.
3. $\{x_1 \mapsto f[x_1]\}\{x_1 \mapsto y_1\}\ g[y_1, f[x_1]]$.

4. $\{x_1 \mapsto y_1\}\{x_1 \mapsto f[x_1]\}\ g[y_1, f[x_1]]$.
5. $\{x_1 \mapsto y_1, y_1 \mapsto f[x_1]\}\ g[y_1, f[x_1]]$.

In the implementation, the lift from σ to σ° gets handled by the following application functions for application of a substitution to a term or a term list.

```
appT :: Subst -> Term -> Term
appT b (Var y)       = appI b y
appT b (Struct n ts) = Struct n (appTs b ts)

appTs :: Subst -> [Term] -> [Term]
appTs = map . appT
```

It is convenient to apply the terminology for substitutions σ also to the corresponding term mappings σ°. Thus, we agree that $\text{Dom}\,(\sigma^\circ) := \text{Dom}\,(\sigma)$ and $\text{Rng}\,(\sigma^\circ) := \text{Rng}\,(\sigma)$.

Note that terms, as defined here, do not contain *variable binders*. As soon as we have variable binders we should make a distinction between bound and free occurrences of variables, and we should make sure that we only substitute for the free occurrences.

If $\sigma : T \to T$ and v is a variable we let σ_v be given by $\sigma_v(w) := w$ if w is the same variable as v, $\sigma_v(w) := \sigma(w)$ otherwise.

If $\sigma : V \to T$ is a substitution then the function $\sigma^\bullet : L \to L$ is defined by structural recursion on the formulas of L, as follows:

$$\begin{array}{rcl}
\sigma^\bullet P[t_1,\ldots,t_n] & := & P[\sigma t_1,\ldots,\sigma t_n] \\
\sigma^\bullet \neg F & := & \neg\sigma^\bullet F \\
\sigma^\bullet(\wedge)[F_1,\ldots,F_n] & := & (\wedge)[\sigma^\bullet F_1,\ldots,\sigma^\bullet F_n] \\
\sigma^\bullet(\vee)[F_1,\ldots,F_n] & := & (\vee)[\sigma^\bullet F_1,\ldots,\sigma^\bullet F_n] \\
\sigma^\bullet(\Rightarrow)F_1F_2 & := & (\Rightarrow)\sigma^\bullet F_1\sigma^\bullet F_2 \\
\sigma^\bullet(\Leftrightarrow)F_1F_2 & := & (\Leftrightarrow)\sigma^\bullet F_1\sigma^\bullet F_2 \\
\sigma^\bullet\forall wF & := & \forall w(\sigma_v)^\bullet F \\
\sigma^\bullet\exists wF & := & \exists w(\sigma_v)^\bullet F
\end{array}$$

Note that the first clause in the definition uses $\sigma^\circ : T \to T$, the term map corresponding to σ. Note also that this definition substitutes only for free occurrences of variables. Here is an example application of the definition:

$$\{x \mapsto g[x,y], y \mapsto z\}\ \forall x P[g[x,y]] = \forall x P[g[x,z]].$$

And here is the corresponding implementation:

```
appF :: Subst -> Form -> Form
appF b (Atom a ts)         = Atom a (appTs b ts)
appF b (Neg form)          = Neg (appF b form)
appF b (Impl form1 form2) = Impl (appF b form1) (appF b form2)
appF b (Equi form1 form2) = Equi (appF b form1) (appF b form2)
appF b (Conj forms)        = Conj (appFs b forms)
appF b (Disj forms)        = Disj (appFs b forms)
appF b (Forall v form)     = Forall v (appF b' form)
                             where b' = filter (\ (x,_) -> x /= v) b
appF b (Exists v form)     = Exists v (appF b' form)
                             where b' = filter (\ (x,_) -> x /= v) b

appFs :: Subst -> [Form] -> [Form]
appFs b = map (appF b)
```

Again, usually we blur the distinction between σ and $\sigma^\bullet$ by omitting the $\bullet$. Thus, if σ is a substitution and E an expression (term or formula) then we write σE for the application of σ° or $\sigma^\bullet$ to E, and we call σE an *instance* of E.

Exercise 6 *The example $\{y \mapsto x\}\forall x R[x,y]$ illustrates a problem that we may encounter when performing a substitution in a quantified formula. The problem is this: the variable y in the original formula is free, but the variable x that replaces it gets bound by the universal quantifier that ranges over x. This is unfortunate, for the formulas $\forall x R[x,y]$ and $\forall z R[z,y]$ intuitively mean the same thing, and $\{y \mapsto x\}\forall z R[z,y]$ does not lead to 'accidental capture' of the newly introduced variable x. Define a notion 't is free for v in F' stating the conditions under which $\{v \mapsto t\}F$ will not lead to accidental capture of variables in t.*

The kind of substitutions we need for reasoning in functional programming, in Prolog, and in automated theorem proving are term substitutions. Let a set of variables V and a language of terms T be given.

It is useful to be able to switch back and forth between term maps (functions in $T \to T$) and substitutions (sets of bindings $v \mapsto t$).

If $\theta = \{v_1 \mapsto \theta v_1, \dots, v_n \mapsto \theta v_n\}$ and $\sigma = \{w_1 \mapsto \sigma w_1, \dots, w_m \mapsto \sigma w_m\}$ are substitution in explicit form, where $\{v_1, \dots, v_n\} = \text{Dom}\,(\theta)$, and $\{w_1, \dots, w_m\} = \text{Dom}\,(\sigma)$, what then is the explicit form of their composition? By the composition we mean the function $\theta^\circ \cdot \sigma : V \to T$ that is the result of applying $\theta^\circ : T \to T$ after $\sigma : V \to T$. To remain close to the prefix notation for functions of the implementation language, we will write substitutions as prefix operators, and read composition of substitutions $\theta\sigma$ as 'θ after σ'.

The explicit form for $\theta \cdot \sigma$ (again we omit the $\circ$) can be obtained by employing the recipe of the following definition:

Definition 1 (Composition of substitution representations) *Let*

$$\theta = [v_1 \mapsto t_1, \dots, v_n \mapsto t_n] \textit{ and } \sigma = [w_1 \mapsto r_1, \dots, w_m \mapsto r_m]$$

be substitution representations. Then $\theta \circ \sigma$ is the result of removing from the sequence

$$[w_1 \mapsto \theta(r_1), \dots, w_m \mapsto \theta(r_m), v_1 \mapsto t_1, \dots, v_n \mapsto t_n]$$

the bindings $w_1 \mapsto \theta(r_i)$ for which $\theta(r_i) = w_i$, and the bindings $v_j \mapsto t_j$ for which $v_j \in \{w_1, \dots, w_m\}$.

Exercise 7 *Prove that this definition gives the correct result.*

Bindings of the form $v \mapsto v$ have to be suppressed from a substitution representation. The function `cleanUp` takes care of this.

```
cleanUp :: Subst -> Subst
cleanUp = filter (\ (x,x') -> x' /= (Var x))
```

Applying the recipe for composition to $\{x \mapsto y\} \cdot \{y \mapsto z\}$ gives $\{y \mapsto z, x \mapsto y\}$, applying it to $\{y \mapsto z\} \cdot \{x \mapsto y\}$ gives $\{x \mapsto z, y \mapsto z\}$. `compose xs ys` implements application of substitution `xs` after substitution `ys`.

```
compose :: Subst -> Subst -> Subst
compose xs ys =
  (cleanUp [ (y,(appT xs y')) | (y,y') <-  ys ])
   ++
  (filter (\ (x,x') -> x `notElem` (dom ys)) xs)
```

To demonstrate this, here are abbreviations for examples.

```
demo1 = compose [(ix,y)] [(iy,x)]
demo2 = compose [(iy,x)] [(ix,y)]
demo3 = compose [(iy,z)] [(ix,y)]
```

This gives:

```
TOTP> demo1
[(x,y)]
TOTP> demo2
[(y,x)]
TOTP> demo3
[(x,z),(y,z)]
```

As we noted already, this illustrates the fact that order of composition matters (substitutions do not commute).

Exercise 8 1. *Write* $\{x \mapsto y\} \cdot \{y \mapsto f[x, y]\}$ *as a substitution in canonical form. Next check your answer with* `compose`.

2. *Write* $\{x \mapsto a[], y \mapsto a[]\} \cdot \{x \mapsto y\}$ *as a substitution in canonical form. Next check your answer with* `compose`.

We extend the terminology for expressions to substitutions. If $\sigma = \rho \cdot \theta$, then we call σ an instance of θ.

Since substitutions are functions, we know immediately that composition of substitutions is associative: if θ, σ, ρ are substitutions, then $(\theta \cdot \sigma) \cdot \rho = \theta \cdot (\sigma \cdot \rho)$.

A substitution $\theta : V \to T$ is called a *renaming* for a set of variables W if $\theta \restriction W$ is injective and the range of $\theta \restriction W$ consists of variables.

In terms of this we define renamings for terms, quantifier-free formulas and variable maps, and renamings *per se*, as follows:

- θ is a renaming for term t if θ is a renaming for $\mathrm{Var}\,(t)$.
- θ is a renaming for quantifier-free formula F if θ is a renaming for $\mathrm{Var}\,(F)$.

- θ is a renaming for variable map σ if θ is a renaming for $\bigcup\{\text{Var}\,(\sigma v) \mid v \in \text{Dom}\,(\sigma)\}$.
- θ is a renaming if θ is a renaming for Dom (θ).

Here is a check whether a substitution is a renaming.

```
isRenaming :: Subst -> Bool
isRenaming sigma = all isVar rngSigma && (nub rngSigma) == rngSigma
  where rngSigma = rng sigma
```

It is easy to see that every renaming θ has an inverse θ^{-1}, with the property that $\theta \cdot \theta^{-1} = \epsilon = \theta^{-1} \cdot \theta$. For if θ is the renaming $\{v_1 \mapsto w_1, \ldots, v_n \mapsto w_n\}$, then $\{w_1 \mapsto v_1, \ldots, w_n \mapsto v_n\}$ is also a renaming, and it has the desired property.

If θ is a renaming for expression E, then θE is called a variant (or: alphabetical variant) of E. For example, $g[x, f[y]]$ is a variant of $g[y, f[x]]$, but $g[x, f[x]]$ is not a variant of $g[y, f[x]]$.

Exercise 9 *Show that if θ is a renaming for E, then there is a renaming σ with $(\sigma \cdot \theta)E = E$.*

Exercise 9 tells us that the relation of *being a variant of* (for expressions) is symmetric. If θ is a renaming for substitution σ, then $\theta \circ \sigma$ is called a variant of σ. The fact that renamings have inverses guarantees that the relation of *being a variant of* (for substitutions) is symmetric. The following exercise turns exercise 9 around:

Exercise 10 *Prove: If $(\sigma \cdot \theta)E = E$, then θ is a renaming for E.*

Exercise 11 *Prove: If E_1, E_2 are terms or quantifier-free formulas, then E_1, E_2 are variants of each other iff they are instances of each other.*

We define a relation $\sqsubseteq$ on the set M of all substitutions (for given sets V and T), as follows. $\theta \sqsubseteq \sigma$ iff there is a substitution ρ with $\theta = \rho \cdot \sigma$. ($\theta \sqsubseteq \sigma$ is sometimes pronounced as: 'θ is less general than σ.')

The relation $\sqsubseteq$ is a pre-order: reflexive and transitive. $\sqsubseteq$ is reflexive because for all θ we have that $\theta = \epsilon \cdot \theta$. $\sqsubseteq$ is transitive because if $\theta = \rho \cdot \sigma$ and $\sigma = \tau \cdot \gamma$ then $\theta = \rho \cdot (\tau \cdot \gamma) = (\rho \cdot \tau) \cdot \gamma$, i.e., $\theta \sqsubseteq \gamma$.

Exercise 12 is similar to Exercise 10, but now for substitutions.

Exercise 12 *Prove: If ρ, θ, σ are substitutions, and $\sigma \cdot \theta \cdot \rho = \rho$, then θ is a renaming for ρ.*

Exercise 13 *Prove that substitutions θ and σ are variants of each other iff they are instances of each other.*

It follows from Exercise 13 that for any substitution σ, the set

$$\{\theta \in M \mid \sigma \sqsubseteq \theta \text{ and } \theta \sqsubseteq \sigma\}$$

is the set which consists of all variants of σ. Thus, using $\sigma \sim \theta$ for $\sigma \sqsubseteq \theta \wedge \theta \sqsubseteq \sigma$, we can form the po-set reflection $(M/_\sim, \preceq)$ of $(M, \sqsubseteq)$, as follows:

$$\begin{aligned} |\sigma| &:= \{\theta \mid \sigma \sim \theta\} \\ |\sigma| \preceq |\rho| &:\equiv \sigma \sqsubseteq \rho. \end{aligned}$$

Thus, the $|\sigma|$ are equivalence classes of substitutions. Intuitively, $|\sigma|$ consists of σ plus all its variants. If $\theta \in |\sigma|$ we say that θ is a representative of $|\sigma|$.

To verify that $\preceq$ is well-defined, we have to show that the relation does not depend on the representatives used in the definition. In other words, if $\sigma \sim \sigma'$, $\rho \sim \rho'$, and $\sigma \sqsubseteq \rho$, then $\sigma' \sqsubseteq \rho'$. For this, note that $\sigma \sim \sigma'$ entails that there is a renaming α with $\sigma' = \alpha \cdot \sigma$, and $\rho \sim \rho'$ entails that there is a renaming β with $\rho = \beta \cdot \rho'$. Now $\sigma \sqsubseteq \rho$ iff there is a θ with $\sigma = \theta \cdot \rho$, so $\sigma' = \alpha \cdot \sigma = \alpha \cdot \theta \cdot \rho = \alpha \cdot \theta \cdot \beta \cdot \rho'$, and it follows that $\sigma' \sqsubseteq \rho'$.

Exercise 14 *Show that $|\epsilon|$ is the set of all renamings.*

2 Unification

If we have two expressions A and B, that each contain variables, then we are interested in the following questions:

- Is there a substitution θ that makes A and B equal?
- How do we find such a substitution in an efficient way?

We introduce some terminology for this. The substitution θ *unifies* expressions A and B if $\theta A = \theta B$. The substitution θ *unifies* two sequences of expressions $(A_1, \ldots, A_n)$ and $(B_1, \ldots, B_n)$ if, for $1 \leq i \leq n$, θ unifies A_i and B_i. Note that unification of pairs of atomic formulas reduces to unification of sequences of terms, for two atoms that start with a different predicate symbol do not unify, and two atoms $P[t_1, \ldots, t_n]$ and $P[s_1, \ldots, s_n]$ unify iff the sequences $[t_1, \ldots, t_n]$ and $[s_1, \ldots, s_n]$ unify.

What we are going to need to apply resolution reasoning (see Section 6) to quantified logic is unification of pairs of atomic formulas. For example, we want to find a substitution that unifies the pair

$$P[x, g[a, z]], P[g[y, z], x].$$

In this example case, such unifying substitutions exist. A possible solution is

$$\{x \mapsto g[a, z], y \mapsto a\}.$$

for applying this substitution gives $P[g[a, z], g[a, z]]$. Another solution is

$$\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\}.$$

In this case, the second solution is an instance of the first, for

$$\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\} \sqsubseteq \{x \mapsto g[a, z], y \mapsto a\},$$

because

$$\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\} = \{z \mapsto b\} \cdot \{x \mapsto g[a, z], y \mapsto a\}.$$

So we see that solution $\{x \mapsto g[a, z], y \mapsto a\}$ is more general than solution $\{x \mapsto g[a, b], y \mapsto a, z \mapsto b\}$.

If a pair of atoms is unifiable, it is useful to try and identify a solution that is as general as possible, for the more general a solution is, the less unnecessary bindings it contains. These considerations motivate the following definition.

Definition 2 *If θ is a unifier for a pair of expressions (a pair of sequences of expressions), then θ is called an mgu (a most general unifier) if $\sigma \sqsubseteq \theta$ for every unifier σ for the pair of expressions (the pair of sequences of expressions).*

In the above example, $\{x \rightarrow g[a, z], y \mapsto a\}$ is an mgu for the pair $P[x, g[a, z]], P[g[y, z], x]$.

Theorem 3 (Unification Theorem) *If a unifier for a pair of sequences of terms exists, then an mgu for that pair exists as well. Moreover, there is an algorithm that produces an mgu for any pair of sequences of terms in case these sequences are unifiable, and otherwise ends with failure.*

Proof. The first part of the theorem follows from the second part, so we will describe the unification algorithm, and prove that it does what it is supposed to do.

We give the algorithm in the form of a Haskell program.

Unification of terms. Three cases:

- Unification of two variables x and y gives the empty substitution if the variables are identical, and otherwise a substitution that binds one variable to the other.
- Unification of x to a non-variable term t fails if x occurs in t, otherwise it yields the binding $\{x \mapsto t\}$.
- Unification of $f\bar{t}$ and $g\bar{r}$ fails if the two variable names are different, otherwise it yields the return of the attempt to do term list unification on $\bar{t}$ and $\bar{r}$.

If unification succeeds, a unit list containing a representation of a most general unifying substitution is returned. Return of the empty list indicates unification failure.

```
unifyTs :: Term -> Term -> [Subst]
unifyTs (Var x)       (Var y)       =
        if x==y then [epsilon] else [[(x,Var y)]]
unifyTs (Var x)       t2            =
        [ [(x,t2)] | x `notElem` varsInTerm t2 ]
unifyTs t1            (Var y)       =
        [ [(y,t1)] | y `notElem` varsInTerm t1 ]
unifyTs (Struct a ts) (Struct b rs) =
        [ u | a==b, u <- unifyTlists ts rs ]
```

Unification of term lists:

- Unification of two empty term lists gives the identity substitution.
- Unification of two term lists of different length fails.
- Unification of two term lists $t_1, \ldots, t_n$ and $r_1, \ldots, r_n$ is the result of trying to compute a substitution $\sigma = \sigma_n \circ \cdots \circ \sigma_1$, where
  - σ_1 is a most general unifier of t_1 and r_1,
  - σ_2 is a most general unifier of $\sigma_1(t_2)$ and $\sigma_1(r_2)$,
  - σ_3 is a most general unifier of $\sigma_2\sigma_1(t_3)$ and $\sigma_2\sigma_1(r_3)$,
  - and so on.

```
unifyTlists :: [Term] -> [Term] -> [Subst]
unifyTlists []     []     = [epsilon]
unifyTlists []     (r:rs) = []
unifyTlists (t:ts) []     = []
unifyTlists (t:ts) (r:rs) =
 [ compose sigma2 sigma1 | sigma1 <- unifyTs t r,
                           sigma2 <- unifyTlists (appTs sigma1 ts)
                                                 (appTs sigma1 rs) ]
```

Our task is to show that these functions do what they are supposed to do: produce a unit list containing an mgu if such an mgu exists, produce the empty list in case unification fails.

The proof consists of a Lemma and two Theorems. The Lemma is needed in Theorem 5. The Lemma establishes a simple property of mgu's. Theorem 6 establishes the result. □

Lemma 4 *If* `sigma1` *is an mgu of* `t1` *and* `s1`*, and* `sigma2` *is an mgu of*

```
[(appT sigma1 t2)..(appT sigma1 tn)],
[(appT sigma1 s2)..(appT sigma1 sn)],
```

then `compose sigma2 sigma1` *is an mgu of* `[t1..tn]` *and* `[s1..sn]`.

Exercise 15 *Prove Lemma 4*

Theorem 5 shows, by induction on the length of term lists, that if `unifyTs t s` does what it is supposed to do, then `unifyTlists` also does what it is supposed to do.

Theorem 5 *Suppose* `unifyTs t s` *yields a unit list containing an mgu of* `t` *and* `s` *if the terms are unifiable, and otherwise yields the empty list. Then* `unifyTlists ts ss` *yields a unit list containing an mgu of* `ts` *and* `ss` *if the lists of terms* `ts` *and* `ss` *are unifiable, and otherwise produces the empty list.*

Proof. If the two lists have different lengths then unification fails. The implementation reflects this, in the cases for `unifyTlists [] (r:rs)` and `unifyTlists (t:ts) []`.

Assume, therefore, that `ts` and `ss` have the same length n. We proceed by induction on n.

Basis $n = 0$, i.e., both `ts` and `ss` are equal to the empty list. In this case the `epsilon` substitution unifies `ts` and `ss`, and this is certainly an mgu.

Induction step $n > 0$. Assume `ts = [t1..tn]` and `ss = [s1..sn]`, with $n > 0$. Then `ts = t1:[t2..tn]` and `ss = s1:[s2..sn]`.

What the algorithm does is:

1. It checks if `t1` and `s1` are unifiable by calling `unifyTs t1 s1`. By the assumption of the theorem, `unifyTs t1 s1` yields a unit list `[sigma1]`, with `sigma1` an mgu of `t1` and `s1` if `t1` and `s1` are unifiable, and yields the empty list otherwise. In the second case, we know that the lists `ts` and `ss` are not unifiable, and indeed, in this case `unifyTlists` will produce the empty list.
2. If `t1` and `s1` have an mgu `sigma1`, then the algorithm tries to unify the lists

   ```
   [(appT sigma1 t2)..(appT sigma1 tn)],
   [(appT sigma1 s2)..(appT sigma1 sn)],
   ```

   i.e., the lists of terms resulting from applying `sigma1` to `[t2..tn]` and `[s2..sn]`. By induction hypothesis we may assume that applying `unifyTlists` to these two lists produces a unit list `[sigma2]`, with `sigma2` an mgu of the lists, if the two lists are unifiable, and the empty list otherwise.
3. If `sigma2` is an mgu of the two lists, then the algorithm returns a unit list containing `compose sigma2 sigma1`. By Lemma 4, `compose sigma2 sigma1` is an mgu of `ts` and `ss`.

□

Theorem 6 clinches the argument. It proceeds by structural induction on terms. The induction hypothesis will allow us to use Theorem 5.

Theorem 6 *The function* `unifyTs t s` *either yields a unit list* `[u]` *or the empty list. In the former case,* `u` *is an mgu of* `t` *and* `s`*. In the latter case,* `t` *and* `s` *are not unifiable.*

Proof. Structural induction on the complexity of `(t,s)`. There are 4 cases.

1. `t` = `Var x`, `s` = `Var y`. In this case, if `x` = `y`, then the `epsilon` substitution is surely an mgu of `t` and `s`. This is what the algorithm yields. If `x` ≠ `y`, then the substitution `[(x,Var y)]` is an mgu of `x` and `y`. For suppose `sigma x` = `sigma y`. Then `sigma x` = `(compose sigma [(x,Var y)]) x`, and for all `z` ≠ `x`, `sigma z` = `(compose sigma [(x, Var y)]) z`. So `sigma` = `compose sigma [(x,Var y)]`.

2. `t` = `Var x`, `s` not a variable. If `x` ∉ `varsIn s`, then `[(x,s)]` is an mgu of `t` and `s`. For if `sigma x` = `sigma s`, then `sigma x` = `(compose sigma [(x,s)]) x`, and for all variables `z` ≠ `x`, `sigma z` = `(compose sigma [(x,s)]) z`. So `sigma` = `compose sigma [(x,s)]`.

3. `s` = `Var x`, `t` not a variable. Similar to case 2.

4. `t` = `Struct a ts` and `s` = `Struct b ss`. Then `t` and `s` are unifiable iff `a` = `b`, and `ts` and `ss` are unifiable. Moreover, `u` is an mgu of `t` and `s` iff `a` = `b` and `u` is an mgu of `ts` and `ss`.

By the induction hypothesis, we may assume for all subterms `t'` and `s'` of `t` and `s` that `unifyTs t' s'` yields the empty list if `t'` and `s'` do not unify, and a unit list `[u]`, with `u` an mgu of `t'` and `s'` otherwise. This means the condition of Theorem 5 is fulfilled, and it follows that `unifyTlists ts ss` yields `[u]`, with `u` an mgu of `ts` and `ss`, if the term lists `ts` and `ss` unify, and `unifyTlists ts ss` yields the empty list if the term lists do not unify.

This establishes the Theorem. □

The following code is useful to play around with the unification algorithm:

```
unif :: Term -> Term -> Subst
unif tm1 tm2 = case unifyTs tm1 tm2 of
   []  -> error "terms do not unify"
   [s] -> s
```

Some output of this:

```
TOTP> unif x y
[(x,y)]
TOTP> unif x (t[x,y])

Program error: terms do not unify

TOTP> unif x (t[y,z])
[(x,t[y,z])]
TOTP> unif (t[x,a]) (t[y,x])
[(x,a),(y,a)]
```

3 Skolemization

For an efficient use of the unification technique from Section 2 in the context of predicate logic, it is common practice to transform the formulas into a more convenient format.

As a crucial step, we will transform the formulas of predicate logic into equivalent quantifier free formulas. For this, we will replace each existential quantification by a so-called skolem term. This process is called skolemization.

Exercise 16 *Below, we will assume that the formulas we start out with do not contain equivalences and implications. Implement a translation function* `transl :: Form -> Form` *that translates every formula into an equivalent formula without occurrences of the constructors* `Impl` *and* `Equi`.

An occurrence of $\exists vF$ in a formula expresses existential quantification if it is within the scope of an even number of negations. An occurrence of $\forall vF$ in a formula expresses existential quantification if it is within the scope of an odd number of negations. Skolemization replaces the quantifiers that express existential quantification. The skolem term for an $\exists vF$ in a positive context (within the scope of an even number of negations) will depend on all universal quantifiers that have scope over $\exists vF$, and similarly for the skolem term for a $\forall vF$ in a negative context (within the scope of an odd number of negations).

Exercise 17 *Replace as many of the quantifiers as you can by appropriate skolem terms.*

1. $\forall x \exists y Rxy$.
2. $\neg \forall x \exists y Rxy$.
3. $\forall x \forall y (\exists z Rxz \wedge Rzy \Rightarrow Rxz)$.
4. $\forall x \forall y (Rxy \Rightarrow \exists z \exists w Szw)$.

In the implementation, we will maintain a list of identifiers for the variables of the outscoping universal quantifiers. A skolem term is constructed from an identifier list and an `Int` as follows. The identifiers represent all the universal parameters that the skolem term depends on.

```
skolem ::  Int -> [Id] -> Term
skolem k is = Struct (Id "sk" [k]) [ (Var x) | x <- is ]
```

An example application: "make a skolem function with index 5 depending on variables x, y, z":

```
TOTP> skolem 5 [xi,yi,zi]
sk_5[x,y,z]
```

To see whether a variable occurrence v in a formula is universally quantified, we not only have to know whether v is bound by $\forall v$ or $\exists v$, but also whether that quantifier is in the scope of an even or an odd number of negations. An occurrence v is universally quantified if

- v is bound by $\forall v$, and $\forall v$ is in the scope of an even number of negations (the subformula $\forall vF$ has positive polarity in the whole formula), or
- v is bound by $\exists v$, and $\exists v$ is in the scope of an odd number of negations (the subformula $\exists vF$ has negative polarity in the whole formula).

The skolemize function `sk` computes by a call to `skf`, an auxiliary function that has a list argument for the current list of wide scope quantifier indices and a Boolean argument to indicate the current polarity, and that passes a parameter for skolem indices.

```
sk :: Form -> Form
sk f = fst (skf f [] True 0)
```

Arguments of `skf`:

1. the first argument is the current formula to be put in skolemized form,
2. the second argument is the list of identifiers for universal quantifiers that have scope over the current formula (these are needed as parameters for the next skolem term),
3. the third argument is the polarity of the current context (`True` for positive, `False` for negative),
4. the fourth argument is the next available `Int` for a skolem identifier (this is needed to ensure that skolem terms for different existentially quantified variables are different).

Note that the code of `skf` uses application of a substitution to a formula.

```
skf :: Form -> [Id] -> Bool -> Int -> (Form,Int)
skf (Atom n ts) ixs pol k = ((Atom n ts),k)
skf (Conj fs) ixs pol k = ((Conj fs'),j)
      where (fs',j) = skfs fs ixs pol k
skf (Disj fs) ixs pol k = ((Disj fs'),j)
      where (fs',j) = skfs fs ixs pol k
skf (Forall x f) ixs True k = ((Forall x f'),j)
     where (f',j) = skf f ixs' True k
           ixs'   = insert x ixs
skf (Forall x f) ixs False k = skf (appF b f) ixs False (k+1)
     where b = [(x,(skolem k ixs))]
skf (Exists x f) ixs True k = skf (appF b f) ixs True (k+1)
     where b = [(x,(skolem k ixs))]
skf (Exists x f) ixs False k = ((Exists x f'),j)
     where (f',j) = skf f ixs' False k
           ixs'   = insert x ixs
skf (Neg f) ixs pol k = ((Neg f'),j)
     where (f',j) = skf f ixs (not pol) k
```

`skfs` puts lists of formulas in skolemized form. Same arguments as `skf`.

```
skfs :: [Form] -> [Id] -> Bool -> Int -> ([Form],Int)
skfs []      _   _   k  = ([],k)
skfs (f:fs) ixs pol k  = ((f':fs'),j)
   where
   (f', j1) = skf  f  ixs pol k
   (fs',j)  = skfs fs ixs pol j1
```

Some example formulas and their skolemized forms. First some relation symbols:

```
p      = Atom (Id "p" [])
q      = Atom (Id "q" [])
r      = Atom (Id "r" [])
```

Then some examples of relational properties expressed in predicate logic.

```
refl   = Forall ix (r [x,x])
irrefl = Forall ix (Neg (r [x,x]))
trans  = Forall ix (Forall iy (Forall iz
         (Disj [Neg (r [x,y]),Neg (r [y,z]),r [x,z]])))
ctrans = Forall ix (Forall iy (Forall iz
         (Disj [r [x,y], r [y,z],Neg (r [x,z])])))
symm   = Forall ix (Forall iy
         (Disj [Neg  (r [x,y]), r [y,x]]))
asymm  = Forall ix (Forall iy
         (Disj [Neg  (r [x,y]), Neg (r [y,x])]))
serial = Forall ix (Exists iy (r [x,y]))
serial1 = Forall ix (Forall iy (Exists iz (r [x,y,z])))
serial2 = Forall ix (Exists iy (Exists iz (r [x,y,z])))
relprop1 = Disj [(Neg asymm),irrefl]
relprop2 = Disj [(Neg trans),(Neg irrefl),asymm]
relprop3 = Disj [(Neg trans),(Neg symm),(Neg serial),refl]
```

And some example skolemizations:

```
TOTP> serial
Ax Ey r[x,y]
TOTP> serial1
Ax Ay Ez r[x,y,z]
TOTP> serial2
Ax Ey Ez r[x,y,z]
TOTP> sk serial
Ax r[x,sk_0[x]]
TOTP> sk serial1
Ax Ay r[x,y,sk_0[x,y]]
TOTP> sk serial2
Ax r[x,sk_0[x],sk_1[x]]
TOTP> sk (Neg serial)
~Ey r[sk_0,y]
TOTP> sk (Neg serial1)
~Ez r[sk_0,sk_1,z]
```

```
TOTP> sk (Neg serial2)
~Ey Ez r[sk_0,y,z]
TOTP> sk (Disj [serial,serial1,serial2])
disj[Ax r[x,sk_0[x]],Ax Ay r[x,y,sk_1[x,y]],Ax r[x,sk_2[x],sk_3[x]]]
TOTP> sk (Neg (Disj [serial,serial1,serial2]))
~disj[Ey r[sk_0,y],Ez r[sk_1,sk_2,z],Ey Ez r[sk_3,y,z]]
```

4 Digression: Conversion to Prenex Form

A formula of first order logic is in *prenex form* or *prenex normal form* if it written as a sequence of quantifiers followed by a quantifier free matrix. Every FOL formula is equivalent to a formula in prenex form.

To make the conversion process go smoothly, assume that every bound variable v in a formula is bound by the same quantifier (this rules out $\forall x Px \wedge \exists x Qx$), and that variables that occur free in a formula do not occur bound in the same formula (this rules out $Px \wedge \exists x Qx$). Call a formula that satisfies these properties *safe*.

Exercise 18 *Write a Haskell function for checking whether a FOL formula is safe.*

Exercise 19 *Write a Haskell function that transforms a FOL formula into an equivalent safe formula.*

To put a formula in prenex form, use the following equivalences:

$$\begin{aligned}
\neg\exists v\varphi &\Leftrightarrow \forall v\neg\varphi\\
\neg\forall v\varphi &\Leftrightarrow \exists v\neg\varphi\\
(\forall v\varphi)\wedge\psi &\Leftrightarrow \forall v(\varphi\wedge\psi)\\
(\forall v\varphi)\vee\psi &\Leftrightarrow \forall v(\varphi\vee\psi)\\
(\exists v\varphi)\wedge\psi &\Leftrightarrow \exists v(\varphi\wedge\psi)\\
(\exists v\varphi)\vee\psi &\Leftrightarrow \exists v(\varphi\vee\psi)
\end{aligned}$$

and the De Morgan laws. Note that the conversions that extend the scope of a quantifier over $\vee$ or $\wedge$ are only valid on the assumption that the formula is safe.

Exercise 20 *Write a Haskell function for converting a (safe) FOL formula to prenex form.*

Exercise 21 *Skolemization and conversion to prenex form are both meaning preserving. Still it is not a good idea to perform skolemization after conversion to prenex form. Why not?*

As a matter of fact, there is no need for conversion to prenex form after skolemization. If we assume that free variables in a formula are universally quantified, then after a formula is skolemized we can just remove all the quantifier occurrences $\forall v$ or $\exists v$. Since the variables are interpreted universally anyway, this does not change the meaning of the formula.

The following function can be used to prune the quantifier occurrences from an (arrow-free) formula in prenex form:

```
prune :: Form -> Form
prune f@(Atom _ _) =  f
prune (Neg f) = Neg (prune f)
prune (Conj fs) = Conj (map prune fs)
prune (Disj fs) = Disj (map prune fs)
prune (Forall _ f) = prune f
prune (Exists _ f) = prune f
```

Exercise 22 *Rewrite the function for skolemization so that it prunes quantifier occurrences on the fly. Check the result by testing whether the new function defines the same transformation as* `prune . sk`.

5 From Formulas to Clauses

Literals and Clauses A *literal* is a atomic formula or its negation. A *clause* is a disjunction of literals. It is customary to write a clause

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B_1 \vee \ldots \vee B_m,$$

where the A_i and the B_j are atomic formulas, as

$$[A_1, \ldots, A_n] \Rightarrow [B_1, \ldots, B_m].$$

All the variables should be taken as universally quantified. Thus, e.g., $[Rxy, Ryz] \Rightarrow [Rxz]$ is the clause that expresses transitivity of R.

Exercise 23 *Translate back into standard predicate logical formulas:*

1. $[] \Rightarrow [Px, Qxy]$.

2. $[Px, Qxy] \Rightarrow []$.
3. $[] \Rightarrow []$.

Blurring the distinction between atomic formulas and terms, we may implement clauses as follows:

```
data Cl = Cl [Term] [Term] deriving (Eq,Ord,Show)
```

The first list holds the negative terms, the second list the positive terms.

Application of a substitution to a clause or a list of clauses:

```
appCl :: Subst -> Cl -> Cl
appCl sigma (Cl neg pos) = Cl (appTs sigma neg) (appTs sigma pos)

appCls :: Subst -> [Cl] -> [Cl]
appCls b = map (appCl b)
```

Variables in a clause:

```
varsInClause :: Cl -> [Id]
varsInClause (Cl neg pos) = nub (varsInTerms neg ++ varsInTerms pos)
```

From Quantifier Free Form to Clause Form Function `fuseLists` will be used to keep the literals in the clauses ordered.

```
fuseLists :: Ord a => [a] -> [a] -> [a]
fuseLists [] ys = ys
fuseLists xs [] = xs
fuseLists (x:xs) (y:ys) | x < y  = x:(fuseLists xs (y:ys))
                        | x == y = x:(fuseLists xs ys)
                        | x > y  = y:(fuseLists (x:xs) ys)
```

Disjunction distribution: a list of clauses (disjunctions) gets transformed into a single clause (disjunction). Note that the empty disjunction (always false) gets represented as the clause `Cl [] []`.

```
disjList :: [Cl] -> [Cl]
disjList []   = [Cl [] []]
disjList [cl] = [cl]
disjList (cl:cls) = map (disj cl) (disjList cls)
  where
  disj (Cl neg pos) (Cl neg' pos') =
        Cl (fuseLists neg neg') (fuseLists pos pos')
```

To put a quantifier free formula in clausal form, one just has to deal with the boolean connectives. A formula translates into a list (conjunction) of clauses.

```
cf :: Form -> [Cl]
cf (Atom n ts) = [Cl [] [Struct n ts]]
cf (Conj fs) = concat (map cf fs)
cf (Disj fs) = disjList (concat (map cf fs))
cf (Neg (Atom n ts)) =  [Cl [Struct n ts] []]
cf (Neg (Conj fs)) = disjList (concat (map (\ f -> cf (Neg f)) fs))
cf (Neg (Disj fs)) = concat (map (\ f -> cf (Neg f)) fs)
cf (Neg (Neg f)) = cf f
```

To put a formula in clausal form, first put it in quantifier free form (skolem form), next clausify the result:

```
clause :: Form -> [Cl]
clause = cf . sk
```

6 Pure Prolog and the Reasoning Engine behind it

A Prolog clause or definite clause is a clause of the form $[A_1, \ldots, A_n] \Rightarrow [A]$, i.e., a clause with a single positive literal. It is customary to write definite clauses in the form $A :- A_1, \ldots, A_n$. In the special case where $n = 0$ we get clauses of the form $A :-$. These are called Prolog facts, and usually written as A. without further ado.

If we represent the atomic predicates as terms, we get the following format (the fixity declaration introduces `:-` as a non-associative infix operator with binding power 6):

```
infix  6 :-
data Dclause =  Term :- [Term] deriving Show
```

A Prolog goal is a sequence of the form $? - A_1, \ldots, A_n$, where the A_i are atomic predicates. Again representing these as terms, we get the following data type:

```
type Goal   =  [Term]
```

Here is an example of a Prolog clause:

$$\mathit{father\_of}(X, Y) :- \mathit{man}(X), \mathit{parent\_of}(X, Y).$$

Our representation of this would be:

```
Struct "father_of" [Var ("X",[]), Var ("Y",[])] :-
  [Struct "man" [Var ("X",[])],
   Struct "parent_of" [Var ("X",[]),Var ("Y",[])]]
```

This clause gives a definition of the relation of fatherhood in terms of the property of being male and the relation of parenthood.

Here is an example with an anonymous variable:

$$\mathit{father}(X) :- \mathit{father\_of}(X, \_).$$

This clause defines the property of being a father in terms of the fatherhood relation: a father is someone who is the the father of someone. Our representation of this would be:

```
father, fatherOf ::  [Term] -> Term
father = Struct (Id "father" [])
fatherOf = Struct (Id "father_of" [])

fatherC :: Dclause
fatherC = father [Var (Id "X" [])]
          :- [fatherOf [Var (Id "X" []),Var (Id "_" [])]]
```

This gets displayed as follows:

```
TOTP> fatherC
father[X] :- [father_of[X,_]]
```

If more anonymous variables occur in a clause, we have to make sure that they all get different representations `_`, `__`, and so on. We do not use the indices for that, as we we need them for a different purpose (see below).

In the syntax of pure Prolog there is no constraint to the effect that a given predicate constant should always have the same arity. The same predicate constant can be used with different numbers of argument terms in the same clause. The following is a correct Prolog clause:

$$\mathit{respect}(X) :- \mathit{respect}(X, X).$$

This defines the property of being a respecter as a thing which respects itself. Our definition of `Goal` handles this correctly. Also, the definition allows Prolog terms like

$$\mathit{father\_of}(\mathit{father\_of}(X))$$

to talk about the paternal grandfather of X,

$$\mathit{father\_of}(\mathit{mother\_of}(X))$$

for talking about the maternal grandfather of X, and terms specifying list patterns, roughly along the lines that we know from Haskell. Prolog uses

$$[[\,]]$$

for the list which has the empty list as its only element,

$$[[\,]|\_]$$

for any list which starts with the empty list, and

$$[a|\_]$$

for any list which has a as its first element. In short, Prolog uses `[t|ts]` where Haskell uses `(t:ts)`. In fact, the notation $[t|ts]$ is an abbreviation for $cons(t, ts)$, just as [] is an abbreviation for *nil*.

The list $[a|\_]$ is represented in our implementation as:

```
Struct "cons" [Struct "a" [],Var ("_",[])]
```

Here are some useful abbreviations:

```
nil :: Term
nil  = Struct (Id "nil" []) []

cons :: [Term] -> Term
cons = Struct (Id "cons" [])
```

The list $[a, b, c]$ is represented in our implementation as:

```
Struct "cons" [Struct "a" [],
   Struct "cons" [Struct "b" [],
      Struct "cons" [Struct "c" [], Struct "nil" []]]]
```

This gets displayed as `cons[a,cons[b,cons[c,nil]]]`.

A Prolog database is a list of predicate definitions, where each predicate definition consists of a list of clauses for a particular predicate (e.g., the predicate `father_of`). We can represent definitions and databases as:

```
type Definition = (Name,[Dclause])
data Database   = Db [Definition] deriving Show
```

Looking up the definition of a particular predicate in a Prolog database is done as follows:

```
dclausesFor            :: Name -> Database -> [Dclause]
dclausesFor a (Db defs) = case dropWhile (\ (n,def) -> n<a) defs of
                          []           -> []
                          ((n,def):_) -> if a==n then def else []
```

The code assumes that the database is ordered by the names of the predicates defined in it. `(\(n,def) -> n<a)` is a specification of a predicate by means of lambda abstraction. It maps a pair `(n,def)` consisting of a name and a definition to the value `False` in case the name precedes the name `a` that we are looking for, and to `True` otherwise.

The function `dropwhile` is predefined in *Prelude.hs* as:

```
dropWhile           :: (a -> Bool) -> [a] -> [a]
dropWhile p []       = []
dropWhile p xs@(x:xs')
         | p x       = dropWhile p xs'
         | otherwise = xs
```

`dropWhile` takes a predicate `p` and a list `xs` and throws away the initial segment of the list of those elements that do not satisfy `p`.

It should be clear that `dclausesFor` finds the definite clauses that define a given predicate if the definition for that predicate is in the database, and returns the empty list otherwise.

Remark. The language of pure Prolog is in fact properly contained in the language of quantified logic, only the formulas appear under a slightly different guise. The following are translation instructions for translating Prolog program facts, program clauses and goal clauses into standard quantified logical notation (irrelevant brackets in conjunctions are omitted):

$$\begin{array}{rcl} A & \rightsquigarrow & \forall x_1 \cdots \forall x_n A \\ A :- A_1, \cdots, A_n & \rightsquigarrow & \forall x_1 \cdots \forall x_n ((A_1 \wedge \cdots \wedge A_n) \rightarrow A) \\ ? - A_1, \cdots, A_n & \rightsquigarrow & \forall x_1 \cdots \forall x_n (\neg A_1 \vee \cdots \vee \neg A_n) \end{array}$$

The list $x_1, \ldots, x_n$ is the list of all variables occurring in the fact or clause. Thus, all Prolog facts, program clauses and goal clauses clauses correspond to closed formulas of quantified logic. ∎

Exercise 24 *Translate the following from Prolog notation to standard quantified logic notation:*

$$father\_of(X, Y) :- man(X), parent\_of(X, Y).$$

$$father(X) :- father\_of(X, \_).$$

Exercise 25 *Translate the following from standard quantified logic notation to Prolog notation. (Hint: which principles do these sentences express? Do they express* different *principles?)*

$$\forall x \forall y \forall z ((Rxy \wedge Ryz) \rightarrow Rxz).$$

$$\forall x \forall z (\exists y (Rxy \wedge Ryz) \rightarrow Rxz).$$

The two main ingredients of the reasoning engine behind Prolog are *resolution*, more in particular SLD resolution, for: selection-rule driven linear resolution for definite clauses, and *unification*.

Propositional resolution is the rule:

$$\frac{C_1 \cup \{P\}, \{\neg P\} \cup C_2}{C_1 \cup C_2}$$

where C_1, C_2 are sets of literals (atomic propositions or negations of atomic propositions) considered as generalized disjunctions.

Exercise 26 *Show that the propositional resolution rule is sound.*

In the case of Prolog we can get by with a more specific version of this. To see why this is so, note that a Prolog goal clause $? - A_1, \ldots, A_n$ is equivalent to a set of literals $\{\neg A_1, \ldots, \neg A_n\}$, where we think of this set as a generalized **disjunction**. A Prolog program clause $B :- B_1, \ldots, B_m$ is equivalent to a clause $\{B, \neg B_1, \ldots, \neg B_m\}$. A Prolog program fact is a special case of this, with $m = 0$. Given a goal clause $\{\neg A_1, \ldots, \neg A_n\}$ and a program clause rule $\{B, \neg B_1, \ldots, \neg B_m$ there is a natural way to to do resolution, by finding an A_i that equals B, so we can do:

$$\frac{\neg A_1, \ldots, \neg A_i, \ldots \neg A_n \quad B :- B_1, \ldots, B_m}{\neg A_1, \ldots, \neg A_{i-1}, \neg B_1, \ldots, \neg B_m, \neg A_{i+1}, \ldots, \neg A_n}\ A_i = B$$

In the propositional case, a question $?-A_1, \ldots, A_n$ is handled by resolving it against a Prolog database (a set of program clauses, or rules), as follows:

$$\cfrac{\cfrac{\cfrac{\mathit{goal}_1 \quad \mathit{rule}_1}{\mathit{goal}_2} \quad \mathit{rule}_2}{\mathit{goal}_3} \quad \mathit{rule}_3}{\cdots}$$

This form of resolution is called *linear resolution* (the name derives from the fact that there is a linear sequence of goal clauses).

In the general case we have to take into account that the predicates may contain variables. So instead of resolving a goal against a rule by selecting a negated atom from the goal list and matching it with the non-negated atom from some program clause by checking whether they are equal, now we attempt to *unify* the two atoms.

Suppose we have a goal $? - A_1, \ldots, A_n$ and a program clause $B : -B_1, \ldots, B_m$ (where B is the head of the clause). The goal is shorthand for $\neg A_1 \vee \ldots \vee \neg A_n$, and the rule for $B \vee \neg B_1 \vee \ldots \vee \neg B_m$. Suppose we want to match A_i against B. This is possible if we can find substitutions α and β for which $\alpha A_i = \beta B$. We can then perform the following general resolution step:

$$\frac{\neg A_1, \ldots, \neg A_i, \ldots, \neg A_n \quad B \leftarrow B_1, \ldots, B_m}{\alpha\neg A_1, \ldots, \alpha\neg A_{i-1}, \beta\neg B_1, \ldots, \beta\neg B_m, \alpha\neg A_{i+1}, \ldots, \alpha\neg A_n}\ \alpha A_i = \beta B$$

Exercise 27 *Show that this reasoning step is sound.*

The result of this reasoning step is a new goal clause.

The key question now becomes: how do we find the substitutions α and β? The answer is, of course, that these are provided by the unification algorithm. We first make sure that the goal $\neg A_1, \ldots, \neg A_i, \ldots \neg A_n$ and the program clause $H \vee \neg B_1 \vee \ldots \vee \neg B_m$ have no variables in common, by applying a renaming σ to the program clause. This is called: standardizing the variables apart.

To standardize variables apart in our implementation, we rename the variables in a term by giving them a new index (depending on the *level* in the current derivation tree; see below).

```
renameVars                  :: Int -> Term -> Term
renameVars level (Var (Id s n))   = Var (Id s [level])
renameVars level (Struct s ts) = Struct s (map (renameVars level) ts)
```

Given a Prolog database, an index that has not been used before for standardizing apart in the current refutation process, and a term representing the currently selected goal-atom from the goal, here is how to find the list of suitably renamed clauses in the database that are candidates for a match:

```
renDclauses                  :: Database -> Int -> Term -> [Dclause]
renDclauses db n (Var _)          = []
renDclauses db n (Struct (Id a _) _) =
             [r tm:-map r tp | (tm:-tp) <- dclausesFor a db]
                               where r = renameVars n
```

In case the goal atom is a variable there are no matches. In case it is a predicate with name `a`, look up the clauses for predicates with that name and rename them by giving them a new index.

Next, we try to unify the selected goal A_i with the head σB of the program clause. If this succeeds, the result is an mgu θ, and we have found $\alpha = \theta$, $\beta = \theta \cdot \sigma$.

$$\frac{\neg A_1, \ldots, \neg A_i, \ldots \neg A_n \quad \sigma B \leftarrow \sigma B_1, \ldots, \sigma B_m}{\theta \neg A_1, \ldots, \theta \neg A_{i-1}, \theta \cdot \sigma \neg B_1, \ldots, \theta \cdot \sigma \neg B_m, \theta \neg A_{i+1}, \ldots, \theta \neg A_n} \ \theta \text{ mgu of } A_i, \sigma B$$

This reasoning step is called an SLD resolution step: S for *selection driven* (a selection rule is employed to select the goal A_i), L for *linear resolution* (the derivation trees are in fact linear sequences), and D for *definite clauses* (indicating that the program clauses are disjunctions with precisely one positive member).

In general, there will be a list of possible matches $B :- B_1, \ldots, B_m$ for a particular goal atom A_i. Each successful match will yield a list of an mgu θ for A_i and B, and a list of clauses $\theta B_1, \ldots, \theta B_m$. Here is a data type for collecting the successful matches:

```
type Alt   = ([Term], Subst)
```

Finding the list of alternatives from a given database, given a fresh index for instantiating apart, and given a goal atom, is done as follows:

```
alts        :: Database -> Int -> Term -> [Alt]
alts db n g = [ (tp,u) | (tm:-tp) <- renDclauses db n g,
                          u       <- unifyTs g tm         ]
```

SLD derivations are sequences of SLD resolution steps. The soundness of SLD resolution follows immediately from the soundness of general resolution.

The selection rule in Prolog is: select the leftmost goal-atom first. Another matter is the selection of the program clause to be matched against the selected goal atom. Here the Prolog search rule is: try the program clauses starting from the first. The search space that the choice of program clauses gives rise to is pictured in so-called SLD derivation trees, where the nodes are labeled by goal clauses, and every successful attempt to match a goal clause against a program clause is indicated by an arc to a daughter (labeled by the number of the program clause and the mgu that was used to effect the match). The leaf nodes are of two kinds:

- nodes labeled with a non-empty goal that cannot be matched against any program clause: these are the failed nodes,
- nodes labeled with the empty goal: these are the success nodes.

An example will hopefully make this clear. Consider the following Prolog predicate definition:

```
/* member(-Item,+List) :- Item occurs in List.  */

member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

The representation for this definition in our Haskell implementation for Prolog clauses is given by:

```
memb :: [Term] -> Term
memb = Struct (Id "member" [])

member :: Definition
member = ("member", [memb [x,cons [x,y]] :- [],
                     memb [x,cons [y,z]] :- [memb [x,z]]])

db = Db [member]
```

The Prolog database db contains just a single definition.

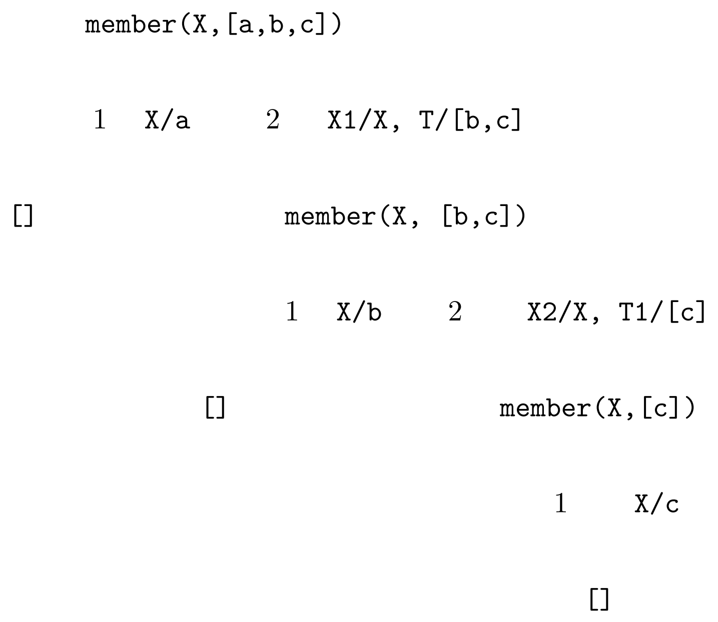


Fig. 1: Finite successful SLD tree.

Figure 6 gives an SLD derivation tree for the goal clause `member(X,[a,b,c])`. Note that before every attempt at finding a match, the variables of the goal and the program clause are instantiated apart. The new program variables are called X_1, X_2 .... All the leaf nodes in this derivation tree are success nodes. This indicates that the refutation of `member(X,[a,b,c])` fails, and indeed every leaf node corresponds with a substitution that makes the goal succeed. These three substitutions for `X` give the three members of `[a,b,c]`.

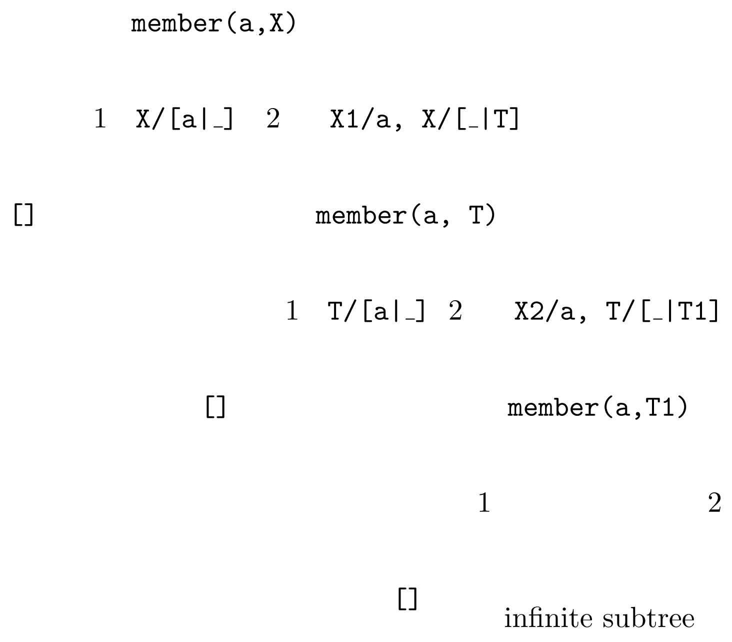


Fig. 2: Infinite successful SLD tree.

Thus, SLD derivation trees picture the Prolog search space for a given goal clause. Note that it does not follow from the definition that an SLD derivation tree should be finite. Indeed, Prolog allows for search processes that are pictured by infinite trees, indicating that the Prolog search process may get into an infinite loop. This is illustrated by the SLD derivation tree for the call to the membership program with a variable in the second position: `member(a,X)`. As the SLD tree in Figure 6 indicates, this Prolog query has infinitely many correct answers: $X = [a|\_], X = [\_, a|\_], X = [\_, \_, a|\_], \ldots$, i.e., the pattern of a list with the element a at the first position, the pattern of a list with the element a at the second position, and so on.

In our implementation, the goal `member (X,[a,b,c])` looks as follows:

```
goal = memb [x, cons [a, cons [b, cons [c,nil]]]]
```

Using the Prolog database from Example 6, we can perform the first resolution step:

```
resolStep = alts db 0 goal
```

The result is the following list of alternatives:

```
TOTP> resolStep
[([],[(x,a),(x_0,a),(y_0,cons[b,cons[c,nil]])]),
 ([member[x_0,z_0]],[(x,x_0),(y_0,a),(z_0,cons[b,cons[c,nil]])])]
```

The first member represents the solution $\{x \mapsto a\}$. The second member yields a substitution $\{x \mapsto x_0, z_0 \mapsto \text{cons}[b, \text{cons}[c, \text{nil}]]\}$ and a new goal `member[x_0,z_0]`.

One way to set up the Prolog reasoning engine is as a stack of triples (σ, G, As), where σ is the current computed substitution, G is the current goal clause, and As is the current list of alternatives, where each alternative gives a different way for continuing the computation to find a solution. If the list of alternatives is empty, the computation fails at that point, and backtracking to a previous backtrack point (σ, G, As) is necessary, by means of the `backtrack` function. If the list of alternatives is non-empty, each alternative is tried in turn by the `choose` function. An empty goal is solved; a non-empty goal gets solved by solving all of its members in turn, by means of the `solve` function.

Here is the code for the Prolog theorem prover (adapted from the Haskell demo Prolog engine):

```
type Stack = [ (Subst, [Term], [Alt]) ]

prove :: Database -> [Term] -> [Subst]
prove db gl = solve 1 epsilon gl []
  where
  solve :: Int -> Subst -> [Term] -> Stack -> [Subst]
  solve n s []     ow = s : backtrack n ow
  solve n s (g:gs) ow = choose n s gs (alts db n (appT s g)) ow

  choose :: Int -> Subst -> [Term] -> [Alt] -> Stack -> [Subst]
  choose n s gs []          ow = backtrack n ow
  choose n s gs ((tp,u):rs) ow =
      solve (n+1) (compose u s) (tp++gs) ((s,gs,rs):ow)

  backtrack :: Int -> Stack -> [Subst]
  backtrack n []              = []
  backtrack n ((s,gs,rs):ow)  = choose (n-1) s gs rs ow
```

For Example 6, this gives the following result:

```
TOTP> prove db [goal]
[[(x,a),(x_1,a),(y_1,cons[b,cons[c,nil]])],
 [(x,b),(y_1,a),(z_1,cons[b,cons[c,nil]]),
  (x_1,b),(x_2,b),(y_2,cons[c,nil])],
 [(x,c),(y_1,a),(z_1,cons[b,cons[c,nil]]),(x_1,c),
  (y_2,b),(z_2,cons[c,nil]),(x_2,c),(x_3,c),(y_3,nil)]]
```

Here is a check how the Prolog engine handles example goal `member(a,X)`:

```
TOTP> take 3 (prove db [memb [a,x]])
[[(x_1,a),(x,cons[a,y_1])],
 [(x_1,a),(x,cons[y_1,cons[a,y_2]]),(x_2,a),(z_1,cons[a,y_2])],
 [(x_1,a),(x,cons[y_1,cons[y_2,cons[a,y_3]]]),
  (x_2,a),(z_1,cons[y_2,cons[a,y_3]]),(x_3,a),(z_2,cons[a,y_3])]]
```

Exercise 28 *Extend the Prolog proof engine with code for handling the cut predicate. The Prolog cut predicate, or "!", eliminates choices in a Prolog derivation tree. The cut goal succeeds whenever it is the current goal, and the derivation tree is trimmed of all other choices on the way back to and including the point in the derivation tree where the cut was introduced into the sequence of goals. E.g., the goal* `member (a,X),!` *would succeed only once.*

7 Resolution Theorem Proving

In the general case, where clauses are not necessarily definite, the resolution rule has to be applied to all possible pairs of clauses that contain complementary literals.

Exercise 29 *Implement resolution theorem proving for arbitrary FOL clause sets. You will need* search *and* resolution with unification*.*

Exercise 30 *Implement resolution theorem proving for arbitrary first order inferences with a premisse set* $\{\varphi_1, \dots, \varphi_n\}$ *and a conclusion* $\psi\}$*.*

Exercise 31 *Validity of inference in FOL is not decidable. How does that reflect in your implementation?*

8 Detecting Inconsistencies with Semantic Tableaux

The method of semantic tableaux was invented in the 1950s by Beth [Bet59] and Hintikka [Hin55] to provide a systematic procedure for detecting logical inconsistencies. If there is an inconsistency in a (finite) set of formulas, we are bound to find it, provided we spell out all the possible states of affairs, while keeping track of all the basic inconsistencies that we encounter, where a basic inconsistency is the presence of a pair of alleged facts $a, \neg a$.

The semantic tableau method proceeds in a step by step fashion, in each step replacing a check of an inconsistency by a simpler check, by decomposition of sentences and distinction of cases. Complex sentences come in two flavours: *conjunctive* compositions and *disjunctive* compositions. The disjunctive compositions are the ones that give rise to a case distinction. Following Smullyan [Smu68], we call *conjunctive* compositions α-sentences and *disjunctive* compositions β-sentences. Here is what their components look like:

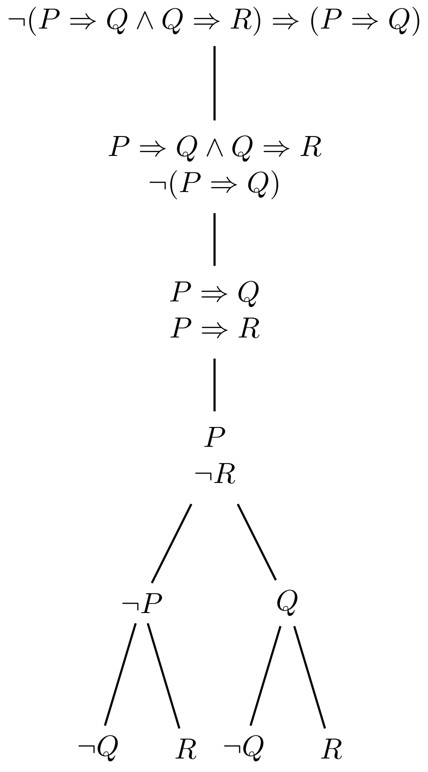
conjunctive			disjunctive		
α	α_1	α_2	β	β_1	β_2
$P \land Q$	P	Q	$\neg(P \land Q)$	$\neg P$	$\neg Q$
$\neg(P \lor Q)$	$\neg P$	$\neg Q$	$P \lor Q$	P	Q
$\neg(P \Rightarrow Q)$	P	$\neg Q$	$P \Rightarrow Q$	$\neg P$	Q
$\neg(P \Leftarrow Q)$	$\neg P$	Q	$P \Leftarrow Q$	P	$\neg Q$

This extends to longer disjunctions and conjunctions in an obvious way. For instance, the sentence $\neg(y_1 \lor b_1 \lor g_1 \lor o_1)$ is an α-sentence, with decomposition $\neg y_1, \neg b_1, \neg g_1, \neg o_1$. Note that $\neg y_1 \land \neg b_1 \land \neg g_1 \land \neg o_1$ is also an α sentence, with the same decomposition.

A tableau for a set of sentences is constructed by applying to sentences from the set the following decomposition rules:

$$\frac{\neg\neg P}{P} \qquad \frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta}{\beta_1 \mid \beta_2}$$

In tableau matters, an example picture to get across the idea often conveys more than a fully spelt out procedure in words. Here is a tableau tree for checking the consistency of $\neg((P \Rightarrow Q \land Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$.

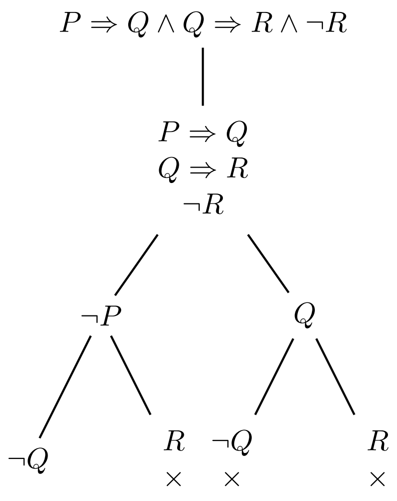

$\neg(P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow Q)$
$P \Rightarrow Q \wedge Q \Rightarrow R$
$\neg(P \Rightarrow Q)$
$P \Rightarrow Q$
$P \Rightarrow R$
P
$\neg R$
$\neg P$
Q
$\neg Q$
R
$\neg Q$
R

The picture shows that first $\neg((P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$ was decomposed in its α_1 and α_2 parts $P \Rightarrow Q \wedge Q \Rightarrow R$ and $\neg(P \Rightarrow R)$. In the next step, the sentence $P \Rightarrow Q \wedge Q \Rightarrow R$ was decomposed into its α_1 and α_2 parts $P \Rightarrow Q$ $Q \Rightarrow Q$. Next, $\neg(P \Rightarrow R)$ was decomposed into its α_1 and α_2 parts P and $\neg R$. At this point the β-sentence $P \Rightarrow Q$ got tackled, causing a split into its β_1 and β_2 components $\neg P$ and Q. Finally, in both branches the β-sentence $Q \Rightarrow R$ got decomposed, causing a further split on both sides.

The tableau shows that the sentence we started out with is inconsistent, for a check of the four branches reveals the pair $P, \neg P$ along the first branch, the pairs $P, \neg P$ and $\neg R, R$ along the second branch, the pair $Q, \neg Q$ along the third branch, and the pair $\neg R, R$ along the fourth branch. This shows that all these avenues are *closed*. In fact, there was no need for the final decomposition step of $Q \Rightarrow R$ on the left hand side, for once a tableau branch is closed, the search for a consistent state of affairs along that avenue is over.

Now if $\neg((P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$ is inconsistent, $(P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$ has to be true no matter what. The tableau method can be viewed as a refutation method: we have tried to refute $(P \Rightarrow Q \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow R)$, but in vain, so we have discovered a truth of logic.

Next consider $P \Rightarrow Q \wedge Q \Rightarrow R \wedge \neg R$. This gives the following tableau; this time we indicate closure of a branch by means of $\times$.



The example indicates that the sentence we started out with is consistent, because in the state of affairs $\neg P, \neg Q, \neg R$ the sentence is true.

Some reflection shows that the sentences that we can harvest from a fully developed open branch in a tableau satisfy a number of simple requirements:

- For no sentence P are both P and $\neg P$ present in the set.
- If a doubly negated sentence $\neg\neg P$ is present in the set, then P is also present.
- If an α-sentence is present in the set, then both its α_1 and α_2 components are present.
- If a β-sentence is present in the set, then either its β_1 or β_2 component is present.

A set of sentences satisfying these requirements is called a Hintikka set.

A propositional state of affairs is called a *Boolean valuation*: if we consider the proposition letters as variables, a Boolean valuation is a function from the set of proposition letters to the truth values **t** and **f**. Every Hintikka set can be taken as an approximation of a valuation: map propositional variable a to **t** if a is in the Hintikka set, to **f** if $\neg a$ is in the Hintikka set. In general, a Hintikka set is not a full specification of a valuation, for there may be basic propositions a that are not decided by the Hintikka set (neither a nor $\neg a$ is in the set).

It is clear that propositional tableaux can be viewed as a systematic procedure for constructing Hintikka sets. If a Hintikka set for a propositional sentence exists, then the tableau method will find it in a finite number of steps. A propositional tableau is *fair* if for any of its branches holds that either the branch is closed, or any α sentence has both its α_1 and its α_2 component on the branch, any β sentence has either its β_1 or its β_2 component on the branch, and any doubly negated sentence $\neg\neg P$ has P on the branch. It takes a finite number of steps to develop a fair tableau for a propositional sentence. Tableaux are a *decision method* for propositional consistency or *propositional satisfiability.*

From Propositional Logic to Predicate Logic If we make the step from propositional logic to predicate logic by allowing predicates and quantification over individuals satisfying those predicates, things get more involved, and more interesting. We now allow sentences of the form 'for all x F holds', or symbolically $\forall xF$, and 'for some x F holds', or symbolically $\exists xF$. The intended meaning is clearly that with respect to a certain universe U a universal sentence $\forall xF$ is true if and only if F remains true, no matter which element d in U we let x refer to. Similarly, an existential sentence $\exists xF$ is true if and only if F remains true for at least one choice of an element d in U that x can refer to. It is useful to have a notation for "F, with x interpreted as d". For this, we use $[x \mapsto d]F$.

The tableau treatment of quantification reflects the way quantifiers are dealt with in mathematical reasoning. If it has been established in the course of a mathematical argument that there exists an object having a certain property P, then it is customary to say: let a be such an object. This boils down to giving one of the things satisfying P the name a. In such cases one always has to make sure that a is not used as a name for anything else: after all, we have not established that anything has P, but just that at least one thing has P. But as long as the baptism does not clash with other naming conventions the switch from $\exists xPx$ to Pa is legitimate. To be on the safe side, one should take a fresh name.

The Smullyan classification of sentences gets extended with universal or γ sentences and existential or δ sentences.

universal		existential	
γ	γ_1	δ	δ_1
$\forall xF$	$[x \mapsto d]F$, d any name	$\exists xF$	$[x \mapsto d]F$, d a new name
$\neg\exists xF$	$[x \mapsto d](\neg F)$, d any name	$\neg\forall xF$	$[x \mapsto d](\neg F)$, d a new name

To facilitate talk about γ and δ sentences and their components, we agree to call the γ_1 component of a certain γ formula, with d as the chosen name, $\gamma(d)$. Similarly, we agree to call the δ_1 component of a certain δ sentence, with d as the new name, $\delta(d)$. Thus, if the γ sentence is $\forall xRxb$, and the chosen name is b, then $\gamma(b)$ indicates the sentence Rbb.

A tableau for a set of sentences of predicate logic is constructed by applying to sentences from the set the following decomposition rules:

$$\frac{\neg\neg F}{F} \quad \frac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \quad \frac{\beta}{\beta_1 \mid \beta_2} \quad \frac{\gamma}{\gamma(d)} \quad \frac{\delta}{\delta(d)}, d \text{ new}$$

It is not difficult to see that the propositional and the γ tableau rules preserve consistency. For the δ rule, we state and prove the consistency as follows:

Proposition 7 *If S is satisfiable, $\delta \in S$, and d is any name that occurs nowhere in S, then $S \cup \{\delta(d)\}$ is satisfiable.*

Proof: satisfiability of S means that there is a universe U and an interpretation I of the predicates from S in U, together with an assignment s that maps the constants in S to elements of U, such that for every $F \in S$ it holds that F_s is true in (U, I). In particular, δ_s is true in (U, I). From this it follows that for at least one $u \in U$, $\delta(u)_s$ is true in (U, I). Now let s' be the assignment that extends s by sending d to u. Then s' is defined for all constants in $S \cup \{\delta(d)\}$, and, since $\delta(d)_{s'}$ equals $\delta(u)_s$, $\delta(d)_{s'}$ is true in (U, I).

If a tableau branch is consistent, i.e., if there is a model that makes all formulas along the branch true, then extending the tableau by means of one of the tableau decomposition rules will result in a consistent tableau. It follows from this that the tableau method is sound:

Theorem 8 (Soundness) *If F is consistent, then any tableau for F will have an open branch.*

The example tableau in Figure 3 establishes that $\neg(\forall x(Px \Rightarrow Qx) \Rightarrow (\forall xPx \Rightarrow \forall xQx))$ is inconsistent, or, in other words, that $\forall x(Px \Rightarrow Qx) \Rightarrow (\forall xPx \Rightarrow \forall xQx)$ is a predicate logical validity.

Note that the object (named) d_1 gets introduced into the tableau by decomposition of the δ sentence $\neg\forall xQx$. This yields $\delta(d_1) = \neg Qd_1$. Next, apply the γ rules, for object d_1, to the sentences $\forall xPx$ and $\forall x(Px \Rightarrow Qx)$.

To start off a tableau for a γ sentence, assume the universe is non-empty, so it contains an object d_1:

***

Since γ formulas impose a standing obligation (for all names d that turn up along a tableau branch, the appropriate $\gamma(d)$ sentence has to be added), predicate logical tableaux can run on indefinitely, as is demonstrated in the tableau for

$$\forall x\exists yRxy \wedge \forall x\forall y\forall z((Rxy \wedge Ryz) \Rightarrow Rxz) \wedge \forall x\neg Rxx$$

Fig. 3: Closed Tableau.

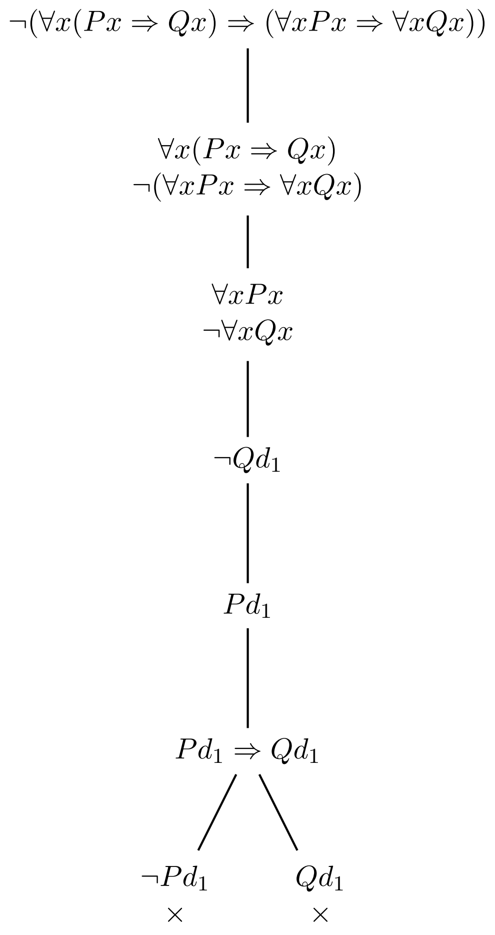

$\neg(\forall x(Px \Rightarrow Qx) \Rightarrow (\forall x Px \Rightarrow \forall x Qx))$
$\forall x(Px \Rightarrow Qx)$
$\neg(\forall x Px \Rightarrow \forall x Qx)$
$\forall x Px$
$\neg\forall x Qx$
$\neg Qd_1$
Pd_1
$Pd_1 \Rightarrow Qd_1$
$\neg Pd_1$
$\times$
Qd_1
$\times$

Fig. 4: Infinite tableau development.

***

in Figure 4. In this example case, this infinite tree generation process is due to the format of the δ rule; a slight relaxation would allow us to re-use d_1 as the object satisfying $\exists y R d_2 y$. This would have resulted in an open branch corresponding to the model $\bullet \longleftrightarrow \bullet$.

But it is easy to see that clever emendations of the tableau rules cannot remedy the situation in general. The extra requirement of asymmetry rules out loops. Thus, the following formula only has infinite models $\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \ldots$

$$\forall x \exists y Rxy \wedge \forall x \forall y \forall z ((Rxy \wedge Ryz) \Rightarrow Rxz) \wedge \forall x \neg Rxx \wedge \forall x \forall y (Rxy \Rightarrow \neg Ryx).$$

A tableau branch corresponding to a model for this sentence is bound to be infinite.

The requirements for a Hintikka set for predicate logic reflect the treatment of universal and existential sentences. A Hintikka set, for a universe U, now is a set of sentences satisfying the following:

- For no sentence F are both F and $\neg F$ present in the set.
- If a doubly negated sentence $\neg\neg F$ is present in the set, then F is also present.
- If an α-sentence is present in the set, then both its α_1 and α_2 components are present.
- If a β-sentence is present in the set, then either its β_1 or β_2 component is present.
- If a γ-sentence is present in the set, then for all $d \in U$, $\gamma(d)$ is in the set.
- If a δ-sentence is present in the set, then for at least one $d \in U$, $\delta(d)$ is present in the set.

Again, fair tableau development should yield Hintikka sets along the open branches. The requirement on γ sentences leads to a crucial difference with the propositional case: there is no longer a guarantee that fair tableaux are finite. As a consequence, tableaux in predicate logic are *not* a decision engine for predicate logical satisfiability.

Crucial for establishing completeness of the tableau method for predicate logic ("if F has a closed tableau then F is not satisfiable") is the following satisfiability lemma for Hintikka sets.

Lemma 9 *Every Hintikka set H for a domain U is satisfiable in U.*

The proof of this is a matter of constructing an appropriate predicate logical model from a Hintikka set. To specify the model, simply give any sentence $Pt_1 \cdots t_n$ the value **t** if $Pt_1 \cdots t_n$ is in H, the value **f** if $Pt_1 \cdots t_n$ is not in H. The terms t_i can range over individual variables of the language, together with names from U. If the language has individual constants, then these should be included in U. Next, show by induction on sentence structure for each sentence F of H that F will get the value $\mathbf{t}$ in this model.

To use this for a completeness proof, we have to establish that open branches in fully developed tableaux yield Hintikka sets. Unfortunately, this is not true without further ado. Since γ rules have to be repeated we can get infinite tableaux. Since tableaux are finitely branching trees, by König's lemma ("Every finitely branching infinite tree has an infinite branch"), an infinite tableau must have an infinite branch. Infinite branches are open, of course, but to make them *fair* we need to develop the tableau according to a strategy guaranteeing a fair treatment of every formula. In particular, not only do all α and β sentences have to be decomposed into their α_1, α_2 or β_1, β_2 components, and all δ sentences have to spawn a $\delta(d)$ for a fresh d, but all γ sentences have to generate $\gamma(d)$ on a branch for all d occurring on the branch. One way to achieve this is to let an application to a γ rule to a branch be followed by the act of putting a copy of the γ formula at the end of the branch [Smu68].

A fair tableau is finished if it cannot be extended by further applications of the fair tableau development procedure. For finished fair tableaux we have that open branches correspond to Hintikka sets. This yields: if F has a finished fair tableau with an open branch, then F is consistent. In other words:

Theorem 10 (Completeness) *If F is a contradiction, then a fair tableau procedure will yield a closed tableau for F after finitely many steps.*

The formulation of the completeness theorem illustrates once more that tableau reasoning is a refutation method. To show that F is a theorem, try to refute $\neg F$ by means of fair tableau development. If this procedure terminates with a closed tableau then F is a theorem. Note again that tableau reasoning for predicate logic is not a decision algorithm: the fair tableau development for $\neg F$ may run forever.

Free Variable Tableaux A problem with the implementation of tableau reasoning for predicate logic is with the efficiency of the treatment of the γ sentences. A γ formula has to yield $\gamma(d)$ for all d along the branch, but which d should be tried first?

The idea of free variable tableaux [Fit96, Hae01] is to postpone this choice by letting a γ formula yield $\gamma(\mathbf{x})$, for a free variable $\mathbf{x}$. This allows the use of *unification* in checking for closure. But one has to be careful, as the following example illustrates:

***

This may look like a closed tableau, but in fact the sentence we started out with is consistent: take universe $\{a, b\}$ and let Pb, Qa give the interpretation of P and Q. The problem is that the free variable $\mathbf{x}$ is a so-called *rigid* variable: it occurs on both sides of a tableau split. The trouble with rigid variables is that they cannot be interpreted universally on a single tableau branch, but have to be read universally on the whole tableau. Thus, closure cannot be checked on single branches, but should be checked globally, for the whole tableau.

Another complication is that introduction of new names in δ rule applications may introduce hidden dependencies on free variables. Consider our earlier example:

$$**$$

This tableau suggests, misleadingly, that $R\mathbf{x}d_1$ will be true irrespective of the reference of $\mathbf{x}$. This is wrong, for it blurs the scope distinction between $\forall x \exists y Rxy$ and $\exists y \forall x Rxy$. In fact, the choice of d_1 is a dependent choice. This has to be made explicit by the use of a so-called skolem function:

By far the easiest way to deal with δ sentences is by prefixing a translation step to skolemized formulas to the tableau procedure, so that the tableau rules do not have to deal with δ sentences at all.

9 The Automation of Tableau Reasoning

To automate tableau reasoning, we need a marriage of tableau development and unification for checking tableau closure. To perform unification on tableaux, let a tableau branch consist of two lists of terms (the terms corresponding to the positive literals, and the terms corresponding to the negative literals), plus a list of pending formulas. A node of a tableau consists of an index in the tableau tree, plus information about the initial segment of the tableau branch up to that node:

```
data Node  = Nd Index [Term] [Term] [Form] deriving Show
```

The tree indexing scheme that we will use is illustrated in the following example tree:

A tableau, expanded to a certain depth, is a list of nodes:

```
type Tableau = [Node]
```

As we work with formulas in skolemized form, there are no δ rule applications. Here is code for distinguishing between α, β and γ formulas:

```
alpha :: Form -> Bool
alpha (Conj _)       = True
alpha (Neg (Disj _)) = True
alpha _              = False

beta :: Form -> Bool
beta (Disj _)       = True
beta (Neg (Conj _)) = True
beta _              = False

gamma :: Form -> Bool
gamma (Forall _ _)       = True
gamma (Neg (Exists _ _)) = True
gamma _                  = False
```

Code for identifying the positive literals, the negative literals, and the double negations:

```
plit, nlit, dneg :: Form -> Bool
plit (Atom n ts)       = True
plit _                 = False
nlit (Neg (Atom n ts)) = True
nlit _                 = False
dneg (Neg (Neg f))     = True
dneg _                 = False
```

Function for converting a literal (an atom or a negation of an atom) to a term:

```
f2t :: Form -> Term
f2t (Atom n ts)       = Struct n ts
f2t (Neg (Atom n ts)) = Struct n ts
```

The components of a (non-literal) formula are given by:

```
components :: Form -> [Form]
components (Conj fs)          = fs
components (Disj fs)          = fs
components (Neg (Conj fs))    = map (\ f -> Neg f) fs
components (Neg (Disj fs))    = map (\ f -> Neg f) fs
components (Neg (Neg f))      = [f]
components (Forall x f)       = [f]
components (Neg (Exists x f)) = [Neg f]
```

For universal sentences, the following function returns the binder:

```
binder :: Form -> Id
binder (Forall x f)       = x
binder (Neg (Exists x f)) = x
```

For universal sentences, the following function allows to strip the list of all universal quantifiers:

```
decompose :: Form -> ([Id],Form)
decompose form = decomp [] form where
  decomp xs f = if gamma f then decomp (xs ++ [x]) f' else (xs,f)
      where x    = binder f
            [f'] = components f
```

Because of the fact that γ formulas are not decomposed tableau expansion can go on forever. An expansion step of a branch looks like this.

It is convenient to prune branches as quickly as possible, by removing nodes that close under *any* substitution.

Note that γ formulas are not removed from the list of pending formulas: if $\forall x F$ is treated, $\forall x F$ gets replaced at the head of the formula list by a renaming of F, and $\forall x F$ gets appended to the formula list. Similarly, if $\neg\exists x F$ is treated, $\neg\exists x F$ gets replaced at the head of the formula list by a renaming of $\neg F$, and $\neg\exists x F$ gets appended to the formula list.

```
step :: Node  -> Tableau
step (Nd i pos neg []) = [Nd i pos neg []]
step (Nd i pos neg (f:fs))
  | plit  f = if elem (f2t f) neg
              then [] else [Nd i ((f2t f):pos) neg fs]
  | nlit  f = if elem (f2t f) pos
              then [] else [Nd i pos ((f2t f):neg) fs]
  | dneg  f = [Nd i pos neg ((components f) ++ fs)]
  | alpha f = [Nd i pos neg ((components f) ++ fs)]
  | beta  f = [(Nd (i++[n]) pos neg (f':fs)) |
                         (f',n)   <- zip (components f) [0..] ]
  | gamma f = [Nd i pos neg  (f':(fs++[f]))]
    where
    (xs,g) = decompose f
    b      = [((Id x j), Var (Id x i)) | (Id x j) <- xs ]
    f'     = appF b g
```

The treatment of γ formulas is a potential source of infinitary behaviour. We can set an arbitrary boundary by keeping track of the number of γ rule applications. For this, we need a version of `step` with a parameter for γ-depth [Fit96].

```
stepD :: Int -> Node -> (Int,Tableau)
stepD k node@(Nd i pos neg []) = (k,[Nd i pos neg []])
stepD k (Nd i pos neg (f:fs))
  | plit  f = if elem (f2t f) neg
              then (k,[]) else (k,[Nd i ((f2t f):pos) neg fs])
  | nlit  f = if elem (f2t f) pos
              then (k,[]) else (k,[Nd i pos ((f2t f):neg) fs])
  | dneg  f = (k,[Nd i pos neg ((components f) ++ fs)])
  | alpha f = (k,[Nd i pos neg ((components f) ++ fs)])
  | beta  f = (k,[(Nd (i++[n]) pos neg (f':fs)) |
                          (f',n)   <- zip (components f) [0..] ])
  | gamma f = (k-1,[Nd i pos neg  (f':(fs++[f]))])
    where
    (xs,g) = decompose f
    b      = [((Id x j), Var (Id x i)) | (Id x j) <- xs ]
    f'     = appF b g
```

A tableau node is fully expanded if its list of pending formulas is empty.

```
expanded :: Node -> Bool
expanded (Nd i pos neg []) = True
expanded  _                = False
```

To expand a tableau up to a given positive γ depth n, apply expansion steps to the first node that needs expansion, until the γ depth gets decreased, next move on with the next node. This ensures that the nodes are treated fairly. Recursively carry out this procedure until the γ depth becomes 0 or the tableau is fully expanded.

```
expand :: Int -> Tableau -> Tableau
expand 0 tableau = tableau
expand _ []      = []
expand n (node:nodes) = if expanded node
                       then (node:(expand n nodes))
                       else if k == n
                       then expand n     (newnodes ++ nodes)
                       else expand (n-1) (nodes ++ newnodes)
    where (k,newnodes) = stepD n node
```

To check a branch for closure at a node, we make use of the fact that the literals are represented as terms, so we can apply **unifyTs** to its lists of positive and negative literals. A node closes when it is possible to unify one of its positive literals against one of its negative literals. The unifying substitutions are needed to instantiate the rest of the tableau, so we collect them in a list.

```
checkN :: Node -> [Subst]
checkN (Nd _ pos neg _) =
   concat [ unifyTs p n | p <- pos, n <- neg ]
```

To check a tableau for closure, we try to close its first branch. For any closing substitution σ that we get, we try to close the σ image of the remaining branches. For this we need functions for applying substitutions to nodes and tableaux.

```
appNd :: Subst -> Node -> Node
appNd b (Nd i pos neg forms) =
  Nd i (appTs b pos) (appTs b neg) (appFs b forms)

appTab :: Subst -> Tableau -> Tableau
appTab = map . appNd
```

The function **checkT** is used in the closure check for a tableau. Note that a tableau consisting of an empty list of nodes counts as closed, because all of its nodes close. A tableau is closed if its list of closing substitutions is non-empty.

```
checkT :: Tableau -> [Subst]
checkT []           = [epsilon]
checkT [node]       = checkN node
checkT (node:nodes) =
  concat [ checkT (appTab s nodes) | s <- checkN node ]
```

No-one can devise a program that decides FOL theorem hood, as Church and Turing discovered in the 1930s. Our refutation engine will expand a formula to a given γ depth, and then check for closure.

The function **`initTab`** creates an initial tableau for a formula.

```
initTab :: Form -> Tableau
initTab form = [Nd [] [] [] [form]]
```

The function **`refuteDepth`** tries to refute a formula by expanding a tableau for it up to a given γ depth.

```
refuteDepth :: Int -> Form -> Bool
refuteDepth k form = checkT tableau /= []
   where tableau = expand k (initTab form)
```

To prove that a formula is a theorem, negate it, put it in skolemized form, and feed it to the refutation engine (for a given γ depth):

```
thm :: Int -> Form -> Bool
thm n = (refuteDepth n) . sk . Neg
```

To check whether a formula is satisfiable, put it in skolemized form, feed it to the refutation engine (for a given γ depth), and negate the answer.

```
sat :: Int -> Form -> Bool
sat n = not . (refuteDepth n) . sk
```

We use the tableau engine to prove that every transitive, symmetric and serial relation is reflexive.

```
formula = Disj [Neg trans, Neg symm, Neg serial, refl]
```

We get:

```
TOTP> thm 10 formula
False
TOTP> thm 20 formula
True
```

This shows that the formula is a theorem of quantified logic.

Exercise 32 *Use the tableau engine to show that every strict partial order is asymmetric.*

Exercise 33 *Use the tableau engine to show that every asymmetric relation is irreflexive.*

10 Further Reading

Excellent textbooks on Prolog are [Bra01, O'K90, SS94]. The logic behind Prolog is explained more fully in Apt [Apt97] and Doets [Doe94]. The classic account of tableau reasoning for first order logic is [Smu68]. An illuminating textbook on theorem proving with first order logic is [Fit96].

References

[Apt97] K.R. Apt. *From Logic Programming to Prolog.* International series in computer science. Prentice Hall, London etc, 1997.

[Bet59] E.W. Beth. *The Foundations of Mathematics.* North Holland, Amsterdam, 1959.

[Bra01] I. Bratko. *Prolog Programming for Artificial Intelligence (3rd edition)*. Addison Wesley, 2001.

[Doe94] H.C. Doets. *From Logic to Logic Programming*. MIT Press, Cambridge, Massachusetts, 1994.

[Fit96] M. Fitting. *First-order Logic and Automated Theorem Proving; Second Edition*. Springer Verlag, Berlin, 1996.

[Hae01] R. Haehnle. Tableaux and related methods. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 2001.

[Hin55] J. Hintikka. Form and content in quantification theory. *Acta Philosophica Fennica*, 8:7–55, 1955.

[O'K90] R.A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[Smu68] R. Smullyan. *First-order logic*. Springer, Berlin, 1968.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog (Second Edition)*. MIT Press, 1994.