HyLoTab — Tableau-based Theorem Proving for Hybrid Logics

Jan van Eijck
CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

February 4, 2002

Abstract

This paper contains the full code of a prototype implementation in Haskell [5], in 'literate programming' style [6], of the tableau-based calculus and proof procedure for hybrid logic presented in [4].

Keywords: Tableau theorem proving for hybrid logic, equality reasoning, model generation, implementation, Haskell, literate programming.

MSC codes: 03B10, 03F03, 68N17, 68T15

1 Introduction

We aim for clarity of exposition rather than efficiency or clever coding.[1] The program consists of the following modules:

Form Describes the representation and handling of formulas. See section 3.

HylotabLex Describes lexical scanning for formulas. Code in the first appendix.

HylotabParse The parsing module, generated by the Haskell parser generator *Happy*. The input for the parser generator is in the second appendix.

Hylotab The module where tableau nodes and tableaux are defined, and where tableau expansion is implemented. See Sections 4 through 15.

Main A module for stand-alone (non-interactive) use of the proof engine. It contains a function `main` that interprets a command line parameter as a file name, reads a formula from the named file, and calls the `sat` function for the formula. Useful for generating compiled code. See Section 16.

The code follows the Haskell'98 standard; it was tested with the Haskell interpreter *Hugs* and with the GHC compiler in both interpret mode *ghci*, and compile mode *ghc*.

[1] and wish more people would do that.

2 Version

This is the first version, written in January 2002. When this gets public, it will get version number 1.00.

3 Datatype Declarations for Nominals and Formulas

Module for formulas:

```
module Form where
import List
```

Identifiers for constant, variable, and tableau parameter nominals are integers, Indices for relations and basic propositions are integers.

```
type Id   = Integer
type Rel  = Integer
type Prop = Integer
```

Nominals can be constants, new parameters (constant nominals added during the tableau development process), or variables. We will need a linear ordering on nominals, so we instruct the Haskell system to derive `Ord` for the datatype.

```
data Nom = C Id | N Id | V Id deriving (Eq,Ord)
```

Declare `Nom` as an instance of the `Show` class:

```
instance Show Nom where
   show (C n) = 'c': show n
   show (V n) = 'x': show n
   show (N n) = 'n': show n
```

Formulas according to

$$\begin{array}{rcl}\varphi & ::= & \top \mid \bot \mid p \mid n \mid x \mid \neg\varphi \mid \bigwedge_i \varphi_i \mid \bigvee_i \varphi_i \mid \varphi \rightarrow \varphi' \mid \\ & & A\varphi \mid E\varphi \mid [\,i\,]\varphi \mid [\,i\,]\breve{}\varphi \mid \langle\, i\,\rangle\varphi \mid \langle\, i\,\rangle\breve{}\varphi \mid @n\varphi \mid \downarrow x.\varphi\end{array}$$

The datatype `Form` defines formulas.

```
data Form =  Bool Bool
           | Prop Prop
           | Nom  Nom
           | Neg  Form
           | Conj [Form]
           | Disj [Form]
           | Impl Form Form
           | A    Form
           | E    Form
           | Box  Rel Form
           | Cbox Rel Form
           | Dia  Rel Form
           | Cdia Rel Form
           | At   Nom Form
           | Down Id Form
     deriving Eq
```

Declare `Form` as an instance of the `Show` class:

```
instance Show Form where
   show (Bool True)         = "T"
   show (Bool False)        = "F"
   show (Prop i)            = 'p':show i
   show (Nom i)             = show i
   show (Neg f)             = '-' : show f
   show (Conj [])           = "T"
   show (Conj [f])          = show f
   show (Conj (f:fs))       =  "(" ++ show f ++ " & " ++ showCtail fs ++ ")"
     where showCtail []     = ""
           showCtail [f]    = show f
           showCtail (f:fs) = show f ++ " & " ++ showCtail fs
   show (Disj [])           = "F"
   show (Disj [f])          = show f
   show (Disj (f:fs))       =  "(" ++ show f ++ " v " ++ showDtail fs ++ ")"
     where showDtail []     = ""
           showDtail [f]    = show f
           showDtail (f:fs) = show f ++ " v " ++ showDtail fs
   show (Impl f1 f2)        = "(" ++ show f1 ++ " -> " ++ show f2 ++ ")"
   show (A f)               = 'A' : show f
   show (E f)               = 'E' : show f
   show (Box name f)        = "[R" ++ show name ++ "]" ++ show f
   show (Cbox name f)       = "[R" ++ show name ++ "~]" ++ show f
   show (Dia name f)        = "<R" ++ show name ++ ">" ++ show f
   show (Cdia name f)       = "<R" ++ show name ++ "~>" ++ show f
   show (At nom f)          = '@': show nom ++ show f
   show (Down i f)          = "Dx" ++ show i ++ "." ++ show f
```

Flatten conjunctions:

```
flattenConj :: [Form] -> [Form]
flattenConj []                = []
flattenConj ((Conj conjs):fs) = flattenConj conjs ++ flattenConj fs
flattenConj ( f          :fs) = f : (flattenConj fs)
```

Flatten disjunctions:

```
flattenDisj :: [Form] -> [Form]
flattenDisj []                = []
flattenDisj ((Disj disjs):fs) = flattenDisj disjs ++ flattenDisj fs
flattenDisj ( f          :fs) = f : (flattenDisj fs)
```

For parsing, it is convenient to have a type for lists of formulas:

```
type Forms = [Form]
```

4 Types for Tableau Development

First we declare the module for tableau development. It imports the standard `List` module, and it imports modules for lexical scanning and parsing. See the appendices for `HylotabLex` and `HylotabParse`.

```
module Hylotab where
import List
import Form
import HylotabLex
import HylotabParse
```

Tableau nodes have indices to point at the next available fresh nominal for the node.

```
type Index    = Integer
```

Tableau nodes have domains: lists of all nominals occurring in the tableau:

```
type Domain   = [Nom]
```

5 Collecting Nominals from Formulas

The following function adds an item to a list, but only in case the item is not yet present:

```
add :: (Eq a) => a -> [a] -> [a]
add x xs = if elem x xs then xs else (x:xs)
```

The following functions collect the constant nominals and the free occurrences of variable nominals from a formula or a formula list.

```
nomsInForm :: Form -> [Nom]
nomsInForm (Bool _)       = []
nomsInForm (Prop _)       = []
nomsInForm (Nom nom)      = [nom]
nomsInForm (Neg form)     = nomsInForm form
nomsInForm (Conj forms)   = nomsInForms forms
nomsInForm (Disj forms)   = nomsInForms forms
nomsInForm (Impl f1 f2)   = nomsInForms [f1,f2]
nomsInForm (A form)       = nomsInForm form
nomsInForm (E form)       = nomsInForm form
nomsInForm (Box _ form)   = nomsInForm form
nomsInForm (Dia _ form)   = nomsInForm form
nomsInForm (Cbox _ form)  = nomsInForm form
nomsInForm (Cdia _ form)  = nomsInForm form
nomsInForm (At nom form)  = add nom (nomsInForm form)
nomsInForm (Down id form) = filter (/= (V id)) (nomsInForm form)

nomsInForms :: [Form] -> [Nom]
nomsInForms = nub . concat. map nomsInForm
```

6 Substitutions

We need to define the result of a substitution of a nominal for a nominal in nominals, in formulas, and in structures defined from those. Represent a substitution as a nominal/nominal pair.

```
type Subst = (Nom,Nom)
```

Application of a substitution to a nominal:

```
appNom :: Subst -> Nom -> Nom
appNom (n,m) nom = if n == nom then m else nom
```

Application of a substitution to a domain. Note that the substitution may identify individuals, so after the substitutition we have to clean up the list with `nub` to remove possible duplicates.

```
appDomain :: Subst -> Domain -> Domain
appDomain = map . appNom
```

Application of a substitution to a list of `(Nom,Nom)` pairs:

```
appNNs :: Subst -> [(Nom,Nom)] -> [(Nom,Nom)]
appNNs b = map (\ (n,m) -> (appNom b n, appNom b m))
```

Application of a substitution to a list of `(Nom,Rel,Nom)` triples:

```
appNRNs :: Subst -> [(Nom,Rel,Nom)] -> [(Nom,Rel,Nom)]
appNRNs b = map (\ (n,r,m) -> (appNom b n, r, appNom b m))
```

Application of a substitution to a list of `(Nom,PropName)` pairs:

```
appNPs :: Subst -> [(Nom,Prop)] -> [(Nom,Prop)]
appNPs b = map (\ (n,name) -> (appNom b n, name))
```

Application of a substitution to a formula or a formula list. Note that in the application of (n, m) to a binder formula $\downarrow v.F$, the occurrences of v inside F are not changed. Substitutions only affect the *free* variables of a formula, and the occurrences of v inside $\downarrow v.F$ are *bound*.

```
appF :: Subst -> Form -> Form
appF b (Bool tv)       = (Bool tv)
appF b (Prop p)        = (Prop p)
appF b (Nom nom)       = (Nom (appNom b nom))
appF b (Neg f)         = Neg (appF b f)
appF b (Conj fs)       = Conj (appFs b fs)
appF b (Disj fs)       = Disj (appFs b fs)
appF b (Impl f1 f2)    = Impl (appF b f1) (appF b f2)
appF b (A f)           = A (appF b f)
appF b (E f)           = E (appF b f)
appF b (Box r f)       = Box r (appF b f)
appF b (Dia r f)       = Dia r (appF b f)
appF b (Cbox r f)      = Cbox r (appF b f)
appF b (Cdia r f)      = Cdia r (appF b f)
appF b (At n f)        = At (appNom b n) (appF b f)
appF (C n,nom) (Down m f) = Down m (appF (C n,nom) f)
appF (N n,nom) (Down m f) = Down m (appF (N n,nom) f)
appF (V n,nom) (Down m f) | n == m    = Down m f
                          | otherwise = Down m (appF (V n,nom) f)

appFs :: Subst -> [Form] -> [Form]
appFs = map . appF
```

Application of a substitution to a list of (Nom,Rel,Form) triples:

```
appNRFs :: Subst -> [(Nom,Rel,Form)] -> [(Nom,Rel,Form)]
appNRFs b = map (\ (n,r,f) -> (appNom b n, r, appF b f))
```

7 Tableau Nodes and Tableaux

A node of a tableau consists of:

1. a tableau index (needed to generate fresh tableau parameters),
2. a domain (all nominals occurring at the node),
3. a list of nominal pairs to represent the $n \not\approx m$ constraints on the node,
4. a list of (n, i, m) triples to represent the nR_im accessibilities on the node,

5. a list of formulas φ to represent the $A\varphi$ formulas that have been applied to all nominals in the domain of the node (the universal constraints of the node),
6. a list of (n, i, φ) triples to represent the $@n[\,i\,]\varphi$ formulas of the node that have been combined with all the nR_im accessibilities on the node (the boxed constraints of the node),
7. a list of (n, i, φ) triples to represent the $@n[\,i\,]\breve{}\varphi$ formulas of the node that have been combined with all the mR_in accessibilities on the node (the converse boxed constraints of the node),
8. a list of (n, i) pairs to represent the positive atom attributions $@np_i$ on the node,
9. a list of (n, i) pairs to represent the negative atom attributions $@n\neg p_i$ on the node,
10. a list of pending formulas at the node (these are the formulas yet to be treated by the proof engine).

```
data Node  = Nd Index
                Domain
                [(Nom,Nom)]
                [(Nom,Rel,Nom)]
                [Form]
                [(Nom,Rel,Form)]
                [(Nom,Rel,Form)]
                [(Nom,Prop)]
                [(Nom,Prop)]
                [Form]
              deriving (Eq,Show)
```

A tableau is a list of its nodes:

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

8 Formula Typology, Formula Decomposition

Code for recognizing α and β formulas:

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

Code for recognizing A and E formulas:

```
isA, isE :: Form -> Bool
isA (A _)              = True
isA (Neg (E _))        = True
isA _                  = False
isE (E _)              = True
isE (Neg (A _))        = True
isE _                  = False
```

Code for recognizing $\Box$, $\breve{\Box}$, $\Diamond$, $\breve{\Diamond}$ formulas:

```
box, cbox, diamond, cdiamond :: Form -> Bool
box (Box _ _)                = True
box (Neg (Dia _ _))          = True
box  _                       = False
cbox (Cbox _ _)              = True
cbox (Neg (Cdia _ _))        = True
cbox  _                      = False
diamond (Dia _ _)            = True
diamond (Neg (Box _ _))      = True
diamond  _                   = False
cdiamond (Cdia _ _)          = True
cdiamond (Neg (Cbox _ _)) = True
cdiamond  _                  = False
```

Code for recognizing $\downarrow$, and @ formulas:

```
down, label :: Form -> Bool
down (Down _ _ )       = True
down (Neg (Down _ _ )) = True
down _                 = False
label (At _ _)         = True
label (Neg (At _ _))   = True
label _                = False
```

Code for identifying the Boolean constants, the positive literals and the negative literals:

```
isBool,plit, nlit :: Form -> Bool
isBool (Bool _)         = True
isBool (Neg (Bool _))   = True
isBool _                = False
plit (Prop _)           = True
plit _                  = False
nlit (Neg (Prop _))     = True
nlit _                  = False
```

Code for identifying the nominals, the negated nominals, the access and converse access formulas and the double negations:

```
isNom, ngNom, isAcc, isCacc, dneg :: Form -> Bool
isNom (Nom _)                      = True
isNom  _                           = False
ngNom (Neg (Nom _))                = True
ngNom _                            = False
isAcc (Dia _ (Nom _))              = True
isAcc (Neg (Box _ (Neg (Nom _))))  = True
isAcc _                            = False
isCacc (Cdia _ (Nom _))            = True
isCacc (Neg (Cbox _ (Neg (Nom _)))) = True
isCacc _                           = False
dneg (Neg (Neg f))                 = True
dneg _                             = False
```

Function for getting the value of a Boolean constant from a formula:

```
getBool :: Form -> Bool
getBool (Bool b)       = b
getBool (Neg (Bool b)) = not b
```

Function for converting a literal (a propositional atom or a negation of a propositional atom) at a nominal n to a pair consisting of the n and the name of the atom.

```
nf2np :: Nom -> Form -> (Nom,Prop)
nf2np nom (Prop name)       = (nom,name)
nf2np nom (Neg (Prop name)) = (nom,name)
```

Function for converting a nominal m or negated nominal $\neg m$, at a nominal n, to the pair (n, m).

```
nf2nn :: Nom -> Form -> (Nom,Nom)
nf2nn n (Nom m)       = (n,m)
nf2nn n (Neg (Nom m)) = (n,m)
```

Function for getting the nominal out of a nominal formula, a negated nominal formula, or an access formula (a formula of the form $\langle i \rangle m$, $\neg[i]\neg m$, $\langle i \rangle\breve{}\, m$ or $\neg[i]\breve{}\,\neg m$):

```
getNom :: Form -> Nom
getNom (Nom nom)                        = nom
getNom (Neg (Nom nom))                  = nom
getNom (Dia _ (Nom nom))                = nom
getNom (Neg (Box _ (Neg (Nom nom))))    = nom
getNom (Cdia _ (Nom nom))               = nom
getNom (Neg (Cbox _ (Neg (Nom nom)))) = nom
```

Function for getting the relation and the target nominal out of a box or diamond formula:

```
getRel :: Form -> Rel
getRel (Dia rel _)        = rel
getRel (Box rel _)        = rel
getRel (Neg (Dia rel _))  = rel
getRel (Neg (Box rel _))  = rel
getRel (Cdia rel _)       = rel
getRel (Cbox rel _)       = rel
getRel (Neg (Cdia rel _)) = rel
getRel (Neg (Cbox rel _)) = rel
```

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

```
components :: Form -> [Form]
components (Conj fs)          = fs
components (Disj fs)          = fs
components (Impl f1 f2)       = [Neg f1,f2]
components (Neg (Conj fs))    = map Neg fs
components (Neg (Disj fs))    = map Neg fs
components (Neg (Impl f1 f2)) = [f1,Neg f2]
components (Neg (Neg f))      = [f]
components (A f)              = [f]
components (Neg (A f))        = [Neg f]
components (E f)              = [f]
components (Neg (E f))        = [Neg f]
components (Box _ f)          = [f]
components (Neg (Box _ f))    = [Neg f]
components (Cbox _ f)         = [f]
components (Neg (Cbox _ f))   = [Neg f]
components (Dia _ f)          = [f]
components (Neg (Dia _ f))    = [Neg f]
components (Cdia _ f)         = [f]
components (Neg (Cdia _ f))   = [Neg f]
components (Down x f)         = [f]
components (Neg (Down x f))   = [Neg f]
components (At nom f)         = [f]
components (Neg (At nom f))   = [Neg f]
```

Located components of a (non-literal) formula:

```
lcomponents :: Nom -> Form -> [Form]
lcomponents nom f = [At nom f' | f' <- components f ]
```

For label formulas, the following function returns the label:

```
getLabel :: Form -> Nom
getLabel (At nom _)       = nom
getLabel (Neg (At nom _)) = nom
```

For binder formulas, the following function returns the binder:

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

Check of a list of formulas for contradiction (not used now; but could be included in the definition of `check` below):

```
checkFs :: [Form] -> Bool
checkFs fs = all (\ (f,f') ->  (f /= (Neg f') && (Neg f) /= f')) fpairs
  where fpairs = [ (f,f') | (At c f) <- fs, (At c' f') <- fs, c == c' ]
```

Checking a node for closure, with closure indicated by return of [].

```
check :: Node -> [Node]
check nd@(Nd i dom neqs accs ufs boxes cboxes pos neg fs) =
  if (checkNN neqs)
     && (checkPN pos neg)
--   && (checkFs fs)
  then [nd] else []
  where checkNN           = all (\(m,n) -> m /= n)
        checkPN poss negs = (intersect poss negs) == []
```

9 Tableau Expansion

While expanding a tableau, it is convenient to prune branches as quickly as possible, by immediately removing closing nodes. Immediate closure is indicated by []. Also, we should take care not to introduce duplicates into the nodes; the function `nub` cleans up lists by removing duplicates.

We now turn to the treatment of an expansion step of a branch (node). This function is at the heart of the program:

```
step :: Node  -> Tableau
```

Applying an expansion step to a node N yields a tableau, i.e., a list of nodes. If the expansion step results in closure of N, we return []. Otherwise we return a non-empty list of open tableau nodes.

If the function is called for a node N with an empty pending formula list, there is nothing left to do, so we return $[N]$.

```
step (Nd i dom neqs accs ufs boxes cboxes pos neg []) =
     [Nd i dom neqs accs ufs boxes cboxes pos neg []]
```

If the list of pending formulas starts with a Boolean constant, then remove it if it is the constant `True`, otherwise close the node:

```
step (Nd i dom neqs accs ufs boxes cboxes pos neg ((At nom f):fs))
  | isBool f = if getBool f
               then [Nd i dom neqs accs ufs boxes cboxes pos neg fs]
               else []
```

The list of pending formulas starts with a propositional literal: check for closure; if the node does not close, then add the literal to the appropriate list.

```
| plit  f = let
              ni   = nf2np nom f
              pos' = add ni pos
            in
  if elem ni neg then []
  else [Nd i dom neqs accs ufs boxes cboxes pos' neg fs]
| nlit  f = let
              ni   = nf2np nom f
              neg' = add ni neg
            in
  if elem ni pos then []
  else [Nd i dom neqs accs ufs boxes cboxes pos neg' fs]
```

The list of pending formulas starts with a nominal. In this case we perform a substitution and check for closure. Note the following:

- Applying a substitution to a list of access relations may result in new access relations, thus destroying the invariant that the box and converse box constraints of the node have been applied for all access relations of the node. To restore that invariant, we have to take care that all box and converse box constraints get applied to the new access relations.
- Applying a substitution to a list of box constraints may result in new box constraints, thus destroying the invariant that the box constraints of the node have been applied for all access relations of the node. To restore that invariant, we have to apply all new box constraints to all access relations of the node.
- Applying a substitution to a list of converse box constraints may result in new converse box constraints, thus destroying the invariant that the converse box constraints of the node have been applied for all access relations of the node. To restore that invariant, we have to apply all new converse box constraints to all access relations of the node.

```
| isNom f =
      let
        k         = getNom f
        m         = min k nom
        n         = max k nom
        dom'      = nub (appDomain (n,m) dom)
        neqs'     = nub (appNNs (n,m) neqs)
        accs'     = nub (appNRNs (n,m) accs)
        ufs'      = nub (appFs (n,m) ufs)
        boxes'    = nub (appNRFs (n,m) boxes)
        newboxes  = boxes' \\ boxes
        cboxes'   = nub (appNRFs (n,m) cboxes)
        newcboxes = cboxes' \\ cboxes
        pos'      = nub (appNPs (n,m) pos)
        neg'      = nub (appNPs (n,m) neg)
        fs'       = nub (appFs  (n,m) fs)
        newaccs   = accs' \\ accs
        bs1       = [ At l g | (k,r,l)   <- newaccs,
                               (k',r',g) <- boxes',
                               k == k', r == r'   ]
        bs2       = [ At l g | (k,r,l)   <- accs',
                               (k',r',g) <- newboxes,
                               k == k', r == r'   ]
        cs1       = [ At k g | (k,r,l)   <- newaccs,
                               (l',r',g) <- cboxes',
                                l == l', r == r'   ]
        cs2       = [ At k g | (k,r,l)   <- accs',
                               (l',r',g) <- newcboxes,
                                l == l', r == r'   ]
        newfs   = nub (fs' ++ bs1 ++ bs2 ++ cs1 ++ cs2)
     in
  check (Nd i dom' neqs' accs' ufs' boxes' cboxes' pos' neg' newfs)
```

The list of pending formulas starts with a negated nominal: check for closure. If the node does not close, add a new inequality $m \not\approx n$ to the inequality list of the node.

```
| ngNom f = if (getNom f) == nom then [] else
        let
          k = getNom f
          m = min k nom
          n = max k nom
          neqs' = add (m,n) neqs
        in
    [Nd i dom neqs' accs ufs boxes cboxes pos neg fs]
```

The list of pending formulas starts with an access formula $@k\Diamond_i n$. Check whether the access relation kR_in is already present at the node. If not, add it, generate the list

$$[@n\varphi \mid A\varphi \in U],$$

where U is the list of universal constraints of the node, the list

$$[@n\varphi \mid @k[\,i\,]\varphi \in B],$$

where B is the set of $\Box$ constraints of the node, and the list

$$[@k\varphi \mid @n[\,i\,]\breve{}\,\varphi \in C],$$

where C is the set of $\Box\breve{}$ constraints of the node, and append these lists to the list of pending formulas.

```
| isAcc f =
        let
          (r,n) = (getRel f, getNom f)
          accs' = add (nom,r,n) accs
          dom'  = add n dom
          fs'   = if elem (nom,r,n) accs then fs
                  else nub (fs ++ us ++ bs ++ cs)
          us    = [ At n g   | g <- ufs             ]
          bs    = [ At n g   | (m,s,g) <- boxes,
                               m == nom, s == r    ]
          cs    = [ At nom g | (m,s,g) <- cboxes,
                               m == n, s == r      ]
        in
    [Nd i dom' neqs accs' ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with a converse access formula $@k\breve{\Diamond}_i n$. Check whether the access relation nR_ik is already present at the node. If not, add it, generate the list

$$[@n\varphi \mid A\varphi \in U],$$

where U is the set of universal constraints of the node, the list

$$[@n\varphi \mid @k[\,i\,]\breve{}\varphi \in C],$$

where C is the set of $\breve{\Box}$ constraints of the node, and the list

$$[@k\varphi \mid @n[\,i\,]\varphi \in B],$$

where B is the set of $\Box$ constraints of the node, and append these lists to the list of pending formulas.

```
| isCacc f =
        let
          (r,n) = (getRel f, getNom f)
          accs' = add (n,r,nom) accs
          dom'  = add n dom
          fs'   = if elem (n,r,nom) accs then fs else
                  nub (fs ++ us ++ bs ++ cs)
          us    = [ At n g     | g <- ufs           ]
          bs    = [ At nom g   | (m,s,g) <- boxes,
                                  m == n, s == r    ]
          cs    = [ At n g     | (m,s,g) <- cboxes,
                                  m == nom, s == r  ]
        in
  [Nd i dom' neqs accs' ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with a double negation: apply the double negation rule.

```
| dneg  f = let
               [g] = lcomponents nom f
               fs' = add g fs
             in
  [Nd i dom neqs accs ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with an α formula: add the components α_i to the node.

```
| alpha f = let
               fs' = nub ((lcomponents nom f) ++ fs)
             in
  [Nd i dom neqs accs ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with a β formula: split the node and add a component β_i to each new branch.

```
  | beta  f = [ Nd i dom neqs accs ufs boxes cboxes pos neg (f':fs) |
                                             f' <- lcomponents nom f  ]
```

The list of pending formulas starts with an A formula $@k\varphi$. Add $\{@m\varphi' \mid m \in D\}$ where D is the domain of the node, to the list of pending formulas, and store φ' as a universal constraint.

```
  | isA   f = let
                newfs  = [ At n g | n <- dom,
                                    g <- components f ]
                fs'    = nub (fs ++ newfs)
                [f']   = components f
                ufs'   = nub (f':ufs)
              in
    [Nd i dom neqs accs ufs' boxes cboxes pos neg fs']
```

The list of pending formulas starts with an E formula $@k\varphi$. Take a fresh nominal n, add it to the domain of the node, add $@n\varphi'$ and $\{@n\psi \mid \psi \in U\}$ to the list of pending formulas.

```
  | isE   f = let
                n      = (N i)
                dom'   = add n dom
                ls     = lcomponents n f
                us     = [ At n g | g <- ufs ]
                fs'    = nub (fs ++ ls ++ us)
              in
    [Nd (succ i) dom' neqs accs ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with a $[\,i\,]$ formula $@k\varphi$. Add the list

$$[@m\varphi' \mid kR_im \in A],$$

where A is the list of access formulas of the node, to the list of pending formulas, and store the $[\,i\,]$ formula as a box constraint. Actually, for convenience, we store (k, i, φ').

```
| box   f = let
            r      = getRel f
            newfs  = [ At n g | (m,s,n) <- accs,
                                g      <- components f,
                                m == nom, s == r           ]
            fs'    = nub (fs ++ newfs)
            boxes' = nub ([(nom,r,g) | g <- components f] ++ boxes)
          in
  [Nd i dom neqs accs ufs boxes' cboxes pos neg fs']
```

The list of pending formulas starts with a $[\,i\,]\breve{}$ formula $@k\varphi$. Add the list

$$[@m\varphi' \mid mR_ik \in A],$$

where A is the list of access formulas of the node, to the list of pending formulas, and store the $[\,i\,]\breve{}$ formula as a converse box constraint. Actually, for convenience we store (k, i, φ').

```
| cbox  f = let
            r       = getRel f
            newfs   = [ At n g | (n,s,m) <- accs,
                                 g      <- components f,
                                 m == nom, s == r           ]
            fs'     = nub (fs ++ newfs)
            cboxes' = nub ([(nom,r,g) | g <- components f] ++ cboxes)
          in
  [Nd i dom neqs accs ufs boxes cboxes' pos neg fs']
```

The list of pending formulas starts with a $\Diamond$ formula: use the node index i to generate a fresh nominal constant n_i, increment the node index, add kR_jn_i to the access list of the node, and put $@n_i\varphi'$, where φ' is the component of the $\Diamond$ formula, on the list of pending formulas. Also, generate appropriate formulas for n_i from the universal constraints and from the box constraints on k, and append them to the list of pending formulas.

```
| diamond f =
          let
            n     = (N i)
            r     = getRel f
            accs' = (nom,r,n):accs
            dom'  = add n dom
            ls    = lcomponents n f
            us    = [ At n g   | g <- ufs        ]
            bs    = [ At n g | (m,s,g) <- boxes,
                                m == nom, s == r  ]
            fs'   = nub (fs ++ ls ++ us ++ bs)
          in
  [Nd (succ i) dom' neqs accs' ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with a $\breve{\Diamond}$ formula: use the node index i to generate a fresh nominal constant n_i, increment the node index, add $n_i R_j k$ to the access list of the node, generate the appropriate formulas for n_i from the converse box constraints of the node, and append them, together with the component of the $\breve{\Diamond}$ formula, to the list of pending formulas.

```
| cdiamond f =
          let
            n      = (N i)
            r      = getRel f
            accs'  = (n,r,nom):accs
            dom'   = add n dom
            ls     = lcomponents n f
            us     = [ At n g | g <- ufs           ]
            cs     = [ At n g | (m,s,g) <- cboxes,
                                m == nom, s == r  ]
            fs'    = nub (fs ++ ls ++ us ++ cs)
           in
  [Nd (succ i) dom' neqs accs' ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with an @ formula $@k@n\varphi$ (or $@k\neg@n\varphi$): add $@n\varphi$ (or $@n\neg\varphi$) to the list of pending formulas.

```
  | label f = let
               fs'  = add f' fs
               n    = getLabel f
               [f'] = lcomponents n f
             in
    [Nd i dom neqs accs ufs boxes cboxes pos neg fs']
```

The list of pending formulas starts with a ↓ formula: add its component to the list of pending formulas, after the appropriate substitution.

```
  | down  f = let
               x   = binder f
               [g] = components f
               f'  = At nom (appF ((V x),nom) g)
               fs' = add f' fs
             in
    [Nd i dom neqs accs ufs boxes cboxes pos neg fs']
```

These are all the possible cases, so this ends the treatment of a single tableau expansion step.

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

```
complete :: Node -> Bool
complete (Nd i dom neqs accs ufs boxes cboxes pos neg []) = True
complete  _                                                = False
```

Expand a tableau in a fair way by performing an expansion step on a (incomplete) node and moving the result to the end of the node list.

```
fullExpand :: Tableau -> Tableau
fullExpand []           = []
fullExpand (node:nodes) = if complete node then node:(fullExpand nodes)
                          else fullExpand (nodes ++ newnodes)
    where newnodes = step node
```

In general, we are not interested in generating all models for a satisfiable formula: one model is enough. This allows for a considerable reduction:

```
expand :: Tableau -> Tableau
expand []           = []
expand (node:nodes) = if complete node then [node]
                      else expand (nodes ++ newnodes)
    where newnodes = step node
```

Tableau development for a given number of steps:

```
develop :: Int -> Form -> Tableau
develop n f = expandN n (initTab f)
  where
    expandN 0 tab        = tab
    expandN n []         = []
    expandN n (node:nodes) = if complete node then [node]
                             else expandN (n-1) (nodes ++ newnodes)
        where newnodes = step node
```

10 Cautious Tableau Expansion

Tableau expansion according to the 'trial and error' versions of the E, $\Diamond$ and $\breve{\Diamond}$ rules, useful for finding minimal models. We replace the `step` function by the following alternative:

```
cautiousStep :: Node -> Tableau
```

Cautious step are like ordinary steps, except for the cases where the formula to be decomposed is a E, $\Diamond$ and $\breve{\Diamond}$ formula.

If the function is called for a node N with an empty pending formula list, again, there is nothing left to do, and we return $[N]$.

```
cautiousStep (Nd i dom neqs accs ufs boxes cboxes pos neg []) =
             [Nd i dom neqs accs ufs boxes cboxes pos neg []]
```

If the list of pending formulas starts with an E formula, we branch to all the possible ways of letting the existential obligation be fulfilled by an existing nominal, and append the result of doing the extension step that introduces a fresh nominal.

```
cautiousStep
  nd@(Nd i dom neqs accs ufs boxes cboxes pos neg ((At nom f):fs))
  | isE   f =
   [ Nd i
        dom
        neqs
        accs
        ufs
        boxes
        cboxes
        pos
        neg
        (nub ((lcomponents n f) ++ fs))
     | n  <- dom
   ]
  ++
  (step nd)
```

If the list of pending formulas starts with a ◇ formula, we branch to all the ways of letting an existing nominal discharge the existential obligation, and append the result of the expansion step that introduces a fresh nominal.

```
  | diamond f =
    let r = getRel f in
   [ Nd i
        dom
        neqs
        (add (nom,r,n) accs)
        ufs
        boxes
        cboxes
        pos
        neg
        (nub ((lcomponents n f)
             ++ fs
             ++ [ At n g | (m,s,g) <- boxes, m == nom, s == r ]))
      | n  <- dom
   ]
  ++
  (step nd)
```

If the list of pending formulas starts with a $\breve{\Diamond}$ formula, we branch to all the ways of letting an existing individual discharge the existential obligation, and append the result of the expansion step that introduces a fresh nominal.

```
| cdiamond f =
  let r = getRel f in
  [ Nd i
       dom
       neqs
       (add (n,r,nom) accs)
       ufs
       boxes
       cboxes
       pos
       neg
       (nub ((lcomponents n f)
            ++ fs
            ++ [ At n g | (m,s,g) <- cboxes, m == nom, s == r ]))
     | n  <- dom
  ]
 ++
 (step nd)
```

In all other cases, we just perform a regular step:

```
| otherwise = step nd
```

11 Theorem Proving

The function `initTab` creates an initial tableau for a formula. The initial tableau for φ just one node, with list of pending formulas $[@m\varphi]$, where m is a fresh nominal. We assume that no nominals of the form `(N i)` appear in the formula. The node index is set to the index of the first fresh constant nominal, i.e., 1.

```
initTab :: Form -> Tableau
initTab form = [Nd 1 dom [] [] [] [] [] [] [] [At nom form]]
  where nom = (N 0)
        dom =  nom : (nomsInForm form)
```

The function `refuteF` tries to refute a formula by expanding a tableau for it, and returning `True` if the tableau closes, `False` if it remains open.

```
refuteF :: Form -> Bool
refuteF form = tableau == []
   where tableau = expand (initTab form)
```

To prove that a formula is a theorem, feed its negation to the refutation function:

```
thm :: Form -> Bool
thm = refuteF . Neg
```

Cautious expansion of a tableau:

```
cautiousExpand :: Tableau -> Tableau
cautiousExpand []           = []
cautiousExpand (node:nodes) = if complete node then [node]
                              else cautiousExpand (nodes ++ newnodes)
    where newnodes = cautiousStep node
```

Cautious tableau development for a given number of steps:

```
cdevelop :: Int -> Form -> Tableau
cdevelop n f = cexpandN n (initTab f)
  where
    cexpandN 0 tab        = tab
    cexpandN n []         = []
    cexpandN n (node:nodes) = if complete node then [node]
                              else cexpandN (n-1) (nodes ++ newnodes)
        where newnodes = cautiousStep node
```

12 Model Generation

```
extract :: Node -> String
extract (Nd i dom _ accs _ _ _ pos neg _) =
  show dom ++ " "
   ++
  concat [ show n ++ "R" ++ show i ++ show m ++ " " | (n,i,m) <- accs ]
   ++
  concat [ "@" ++ show i ++ " p" ++ show p ++ " "   | (i,p) <- pos    ]
   ++
  concat [ "@" ++ show i ++ "-p" ++ show p ++ " "   | (i,p) <- neg    ]
```

Report on an expanded tableau:

```
report :: Tableau -> IO()
report [] = putStr "not satisfiable\n"
report t  = putStr ("satisfiable:\n" ++ models t) where
   models []     = ""
   models (n:ns) = extract n ++ "\n" ++ models ns
```

To check a formula for satisfiability, feed it to the proof engine and report on the resulting tableau.

```
sat :: Form -> IO ()
sat form = report tableau
    where tableau = expand (initTab form)
```

Check a formula for satisfiability, looking for a minimal model:

```
msat :: Form -> IO ()
msat form = report tableau
    where tableau = cautiousExpand (initTab form)
```

13 Frame Properties

Here is a list of pure formulas (formulas without proposition letters) that define frame properties. In fact, *any* pure formula defines a frame condition, and characterizes a class of frames.

transitive	$\downarrow x.\Box\Box\breve{\Diamond}x$
intransitive	$\downarrow x.\Box\Box\breve{\Box}\neg x$
reflexive	$\downarrow x.\Diamond x$
irreflexive	$\downarrow x.\Box\neg x$
symmetric	$\downarrow x.\Box\breve{\Diamond}x$
asymmetric	$\downarrow x.\Box\breve{\Box}\neg x$
serial	$\Diamond\top$
euclidean	$\downarrow x.\Box \downarrow y.@x\Box\Diamond y$
antisymmetric	$\downarrow x.\Box(\Diamond x \to x)$
S4 (refl + trans)	$\downarrow x.(\Diamond x \wedge \Box\Box\breve{\Diamond}x)$
S5 (refl + symm + trans)	$\downarrow x.(\Diamond x \wedge \Box\Diamond x \wedge \Box\Box\breve{\Diamond}x)$

To define functions for these, it is convenient to use a function for converting strings to formulas:

```
str2form :: String -> Form
str2form = parse . lexer
```

Here are the definitions of a number of frame classes that can be extended almost *ad libitum.* Many of these are known from modal logic (see [3] or [7] for the abbreviations), but of course the list also contains examples of properties that cannot be defined in standard modal logic.

```
trans     = str2form "D x [][]<~>x"
k4        = trans
intrans   = str2form "D x [][][~]-x"
refl      = str2form "D x <>x"
kt        = refl
irrefl    = str2form "D x []-x"
symm      = str2form "D x []<>x"
kb        = symm
asymm     = str2form "D x [][~]-x"
s4        = str2form "D x (<>x & [][] <~>x)"
kt4       = s4
s5        = str2form "D x conj (<>x, []<>x, [][] <~>x)"
serial    = str2form "<>T"
kd        = serial
euclid    = str2form "D x [] D y @ x []<>y"
k5        = euclid
kdb       = str2form "(<>T & D x []<~>x)"
kd4       = str2form "(<>T & D x [][]<~>x)"
kd5       = str2form "(<>T & D x [] D y @ x []<>y)"
k45       = str2form "D x([][]<~>x & [] D y @ x []<>y)"
kd45      = str2form "D x conj (<>T, [][]<~>x , [] D y @ x []<>y)"
kb4       = str2form "D x ([]<>x & [][] <~>x)"
ktb       = str2form "D x (<>x & []<>x)"
antisymm  = str2form "D x [](<>x -> x)"
```

14 Generic Satisfiability Checking

If ψ defines a frame property, then any model $\mathcal{M}$ satisfying $A\psi$ will be in the frame class with that property. For suppose that $\mathcal{M}$ does not have the frame property. Then there is a world w with $\mathcal{M}, w \not\models \psi$, hence $\mathcal{M} \not\models A\psi$. Thus, we can build an engine for the frame class of ψ by loading the proof engine with $A\psi$ as a universal constraint. This leads to the following generic satisfiability checker (the first parameter is used to indicate whether we are looking for minimal models or not; the second parameter lists the formulas defining the frame class under consideration):

```
genSat  :: Bool -> [Form] -> Form -> IO()
genSat False [] form = sat form
genSat True  [] form = msat form
genSat False [prop] form = report tableau
  where
  tableau   = expand (inittab form)
  inittab f = [Nd 1 dom [] [] [A prop] [] [] [] [] [At nom form]]
  nom       = (N 0)
  dom       =  nom : (nomsInForms [form,prop])
genSat True [prop] form = report tableau
  where
  tableau   = cautiousExpand (inittab form)
  inittab f = [Nd 1 dom [] [] [A prop] [] [] [] [] [At nom form]]
  nom       = (N 0)
  dom       =  nom : (nomsInForms [form,prop])
genSat False props form = report tableau
  where
  tableau   = expand (inittab form)
  inittab f = [Nd 1 dom [] [] [A (Conj props)] [] [] [] [] [At nom form]]
  nom       = (N 0)
  dom       =  nom : (nomsInForms (form:props))
genSat True props form = report tableau
  where
  tableau   = cautiousExpand (inittab form)
  inittab f = [Nd 1 dom [] [] [A (Conj props)] [] [] [] [] [At nom form]]
  nom       = (N 0)
  dom       =  nom : (nomsInForms (form:props))
```

15 Example Formulas

Some examples to play around with:

$$\Diamond(c \wedge p) \wedge \Diamond(c \wedge p_1) \wedge \Box(\neg p \vee \neg p_1)$$

```
ex1 = str2form "conj (<> (c & p), <>(c & p1), [] (-p v -p1))"
```

$$@x\Diamond\breve{\Box}\neg x \vee @x\breve{\Diamond}\Box\neg x$$

```
ex2 = str2form "(@x<>[~]-x v @x<~>[]-x)"
```

$$\Diamond p \wedge \Diamond \neg p \wedge \Box(p_1 \to c) \wedge \Box p_1$$

```
ex3 = str2form "conj (<>p, <>-p, [](p1 -> c), []p1)"
```

$$\Diamond(c \wedge \Diamond c \wedge \Box\Diamond c)$$

```
ex4 = str2form "<>conj (c, <>c, []<>c)"
```

$$\Diamond\top \wedge \Box\Diamond\top \wedge \Box \downarrow x.\Box\Box\breve{\Diamond} x$$

```
ex5 = str2form "conj (<>T, []<>T, [] down x [][] <~>x)"
```

$$\Diamond\top \wedge \Box\Diamond\top \wedge \downarrow x.\Box\Box\breve{\Diamond} x$$

Note: this will get the `sat` function into a loop (why?). The remedy is in the trial and error versions of the $\Diamond$ and $\breve{\Diamond}$ rules. The `msat` function finds the minimal model.

```
ex6 = str2form "conj (<>T, []<>T, down x [][] <~>x)"
```

The GRID formula from [1]:

$$\begin{array}{lll} @c\neg\Diamond c & \wedge & @c\Diamond\top \\ & \wedge & @c\Box\Box \downarrow x.@c\Diamond x \\ & \wedge & @c\Box \downarrow x.\Box \downarrow x_1.@x\Box \downarrow x_2.@x\Box \downarrow x_3.(@x_1x_2 \vee @x_1x_3 \vee @x_2x_3) \\ & \wedge & @c\Box \downarrow x.\Box\Box \downarrow x_1.@x\Box\Box \downarrow x_2.@x\Box\Box \downarrow x_3.@x\Box\Box \downarrow x_4. \\ & & \qquad (@x_1x_2 \vee @x_1x_3 \vee @x_1x_4 \vee @x_2x_3 \vee @x_2x_4 \vee @x_3x_4) \end{array}$$

```
grid = str2form (
         " conj (@ c -<>c, "
     ++  "   @ c <>T,  "
     ++  "   @ c [][] down x @ c <> x, "
     ++  "   @ c [] down x [] down x1 @ x [] down x2 @ x [] down x3 "
     ++  "      disj ( @ x1 x2,@ x1 x3,@ x2 x3 ), "
     ++  "   @ c [] down x [][] down x1  "
     ++  "             @ x [][] down x2  "
     ++  "             @ x [][] down x3  "
     ++  "             @ x [][] down x4  "
     ++  "                disj ( @ x1 x2, @ x1 x3, @ x1 x4,  "
     ++  "                       @ x2 x3, @ x2 x4, @ x3 x4 ) "
     ++  " )"
                )
```

A simple way to input formulas from files:

```
satIO :: IO ()
satIO = do {
            putStr "Input file: ";
            filename <- getLine;
            fstring <- readFile filename;
            let { form = str2form fstring; };
            putStr ("Formula: " ++ show form ++ "\n");
            sat form
           }
```

```
msatIO :: IO ()
msatIO = do {
             putStr "Input file: ";
             filename <- getLine;
             fstring <- readFile filename;
             let { form = str2form fstring; };
             putStr ("Formula: " ++ show form ++ "\n");
             msat form
            }
```

16 The ‘Main’ Module

This module contains the code for stand-alone use. Module declaration:

```
module Main

where

import Form
import Hylotab
import System
```

Information banner for cases where the *hylotab* command is invoked without command line arguments.

```
showInfo :: IO ()
showInfo = putStrLn ("HyLoTab 1.00: no input file.\n" ++
     "Usage: ‘--help’ option gives basic information.\n")
```

Help on usage:

```
showHelp :: IO ()
showHelp = putStrLn ("Usage: \n" ++
 "  hylotab [SATOPTION] [FRAMEOPTION] ... FILE\n\n" ++
 "  satoption   : --help (print this help information)\n" ++
 "                --min  (search for minimal models)\n" ++
 "  frameoptions: -trans -k4 -intrans -refl -kt -irrefl -symm\n" ++
 "                -kb -asymm -s4 -kt4 -s5 -serial -kd -euclid\n" ++
 "                -k5 -kdb -kd4 -kd5 -k45 -kd45 -kb4 -ktb -antisymm\n\n" ++
 "  Examples of use:\n\n" ++
 "  hylotab -s4 form1           (satisfy form1 in s4 model)\n" ++
 "  hylotab -refl -trans form1  (idem) \n" ++
 "  hylotab --min -kt form1     (satisfy form1 in minimal refl model)\n" ++
 "  hylotab --min -refl form1   (idem)\n\n")
```

List of frame options:

```
frameOptions :: [String]
frameOptions = ["-trans", "-k4", "-intrans", "-refl", "-kt", "-irrefl",
                "-symm", "-kb", "-asymm", "-s4", "-kt4", "-s5",
                "-serial", "-kd", "-euclid", "-k5", "-kdb", "-kd4",
                "-kd5", "-k45", "-kd45", "-kb4", "-ktb", "-antisymm"]
```

Converting an option to the corresponding formula:

```
option2form :: String -> Form
option2form "-trans"    = trans
option2form "-k4"       = k4
option2form "-intrans"  = intrans
option2form "-refl"     = refl
option2form "-kt"       = kt
option2form "-irrefl"   = irrefl
option2form "-symm"     = symm
option2form "-kb"       = kb
option2form "-asymm"    = asymm
option2form "-s4"       = s4
option2form "-kt4"      = kt4
option2form "-s5"       = s5
option2form "-serial"   = serial
option2form "-kd"       = kd
option2form "-euclid"   = euclid
option2form "-k5"       = k5
option2form "-kdb"      = kdb
option2form "-kd4"      = kd4
option2form "-kd5"      = kd5
option2form "-k45"      = k45
option2form "-kd45"     = kd45
option2form "-kb4"      = kb4
option2form "-ktb"      = ktb
option2form "-antisymm" = antisymm
```

Parse the command line arguments. Return of `[]` indicates error. If no error occurs, the return is of the form `[(min,fs,name)]`, where the value of `min` indicates satisfaction in minimal models, the value of `fs` lists the frame conditions, and the value of `name` gives the filename.

```
parseArgs :: [String] -> [(Bool,[Form],String)]
parseArgs args = parseArgs' args (False,[])

parseArgs' :: [String] -> (Bool,[Form]) -> [(Bool,[Form],String)]
parseArgs' [] (min,flist) = []
parseArgs' [name] (min,flist) = if (head name == '-') then []
                                else [(min,flist,name)]
parseArgs' (arg:args) (min,flist)
  | arg == "--help"       = []
  | arg == "--min"        = parseArgs' args (True,flist)
  | elem arg frameOptions = parseArgs' args (min,((option2form arg):flist))
  | otherwise             = []
```

The function `main` parses the commandline and, if all goes well, calls the generic satisfaction function in the appropriate way for the formula read from the named file.

```
main :: IO ()
main = do {
   args <- getArgs;
   if null args then showInfo
           else if parseArgs args == [] then showHelp
           else let [(min,fs,name)] = parseArgs args in
                     do {
                          fstring <- readFile name;
                          let { form = str2form fstring;  };
                          putStr ("Formula: " ++ show form ++ "\n");
                          genSat min fs form
                        }
          }
```

17 Known Bugs, Future Work

Further testing of the code is (always) necessary. Apart from that, the following are on the agenda:

- The parser does not produce useful error messages. Implementing a less primitive version that returns error messages with line numbers is future work. It is probably most useful to make this a joint effort with *HyLoRes*. Unfortunately, the *HyLoRes* parser does a lot of preprocessing, while we have made it our policy to remain faithful to syntactic form.

- Writing of a nice WEB interface to *HyLoTab*.
- Graphical presentation of generated Kripke models, e.g., by incorporation of features from Lex Hendriks' *Akka* (see `http://turing.wins.uva.nl/~lhendrik/AkkaStart.html`) and Marc Pauly's PDL model editor based on it (see `http://www.cwi.nl/~pauly/Applets/ModEd.html`).
- The node compression algorith from the paper is not yet in the implementation.
- Detailed comparison with *HyLoRes* [2], also written in Haskell.

Acknowledgement Many thanks to the Dynamo team (Wim Berkelmans, Balder ten Cate, Juan Heguiabehere, Breanndán Ó Nualláin) and to Carlos Areces, Patrick Blackburn and Maarten Marx, for useful comments, fruitful discussion and bug chasing help.

References

[1] Areces, C., Blackburn, P., and Marx, M. Hybrid logics: Characterization, interpolation and complexity. *Journal of Symbolic Logic* (2001).

[2] Areces, C., and Heguiabehere, J. Hylores: Direct resolution for hybrid logics. In *Proceedings of Methods for Modalities 2* (Amsterdam, The Netherlands, November 2001), C. Areces and M. de Rijke, Eds.

[3] Chellas, B. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[4] Eijck, J. v. Constraint tableaux for hybrid logics. Manuscript, CWI, Amsterdam, 2002.

[5] Jones, S. P., Hughes, J., et al. Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

[6] Knuth, D. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[7] Popkorn, S. *First Steps in Modal Logic*. Cambridge University Press, 1994.

Appendix 1: Lexical Scanning

```
module HylotabLex

where

import Prelude
import Char
```

Names for all tokens:

```
data Token =
    TokenAt1          | TokenAt2          | TokenDot          |
    TokenImpl         | TokenDimp         | TokenNeg          |
    TokenAnd          | TokenOr           |
    TokenConj         | TokenDisj         |
    TokenProp String  | TokenCst String   | TokenVar String   |
    TokenTrue         | TokenFalse        |
    TokenA            | TokenE            |
    TokenBox String   | TokenDia String   |
    TokenCbox String  | TokenCdia String  | TokenComma        |
    TokenBnd          | TokenOB           | TokenCB
                                                  deriving Show
```

The lexer transforms an input string into a list of tokens.

```
lexer :: String -> [Token]
lexer [] = []
lexer ('<':'-':'>':cs) = TokenDimp : lexer cs
lexer ('-':'>':cs) = TokenImpl : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
lexer ('A':cs) = TokenA : lexer cs
lexer ('E':cs) = TokenE : lexer cs
lexer ('d':'i':'a':cs) = (TokenDia "0") : lexer cs
lexer ('c':'d':'i':'a':cs) = (TokenCdia "0") : lexer cs
lexer ('<':cs) = lexDia cs
lexer ('b':'o':'x':cs)     = (TokenBox "0") : lexer cs
lexer ('c':'b':'o':'x':cs) = (TokenCbox "0") : lexer cs
lexer ('[':cs) = lexBox cs
lexer ('|':cs) = TokenOr: lexer cs
lexer ('v':'v':cs) = TokenDisj: lexer cs
lexer ('v':cs) = TokenOr: lexer cs
lexer ('&':'&':cs) = TokenConj: lexer cs
lexer ('&':cs) = TokenAnd: lexer cs
lexer ('^':cs) = TokenAnd: lexer cs
lexer ('-':cs) = TokenNeg: lexer cs
lexer (',':cs) = TokenComma: lexer cs
lexer ('.':cs) = TokenDot: lexer cs
lexer ('c':'o':'n':'j':cs) = TokenConj: lexer cs
lexer ('d':'i':'s':'j':cs) = TokenDisj: lexer cs
lexer (':':cs) = TokenAt1: lexer cs
lexer ('@':cs) = TokenAt2: lexer cs
lexer ('!':cs) = TokenBnd: lexer cs
lexer ('D':cs) = TokenBnd: lexer cs
lexer ('d':'o':'w':'n':cs) = TokenBnd: lexer cs
lexer ('{':cs) = lexComment cs 0
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexName (c:cs)
```

Allow nested comments:

```
lexComment :: [Char] -> Integer -> [Token]
lexComment (c:cs) n | c == '}'  = case n <= 0  of
                                    True  -> lexer cs
                                    False -> lexComment cs (n-1)
                    | c == '{'  = lexComment cs (n+1)
                    | otherwise = lexComment cs n
```

Recognize boxes, while distinguishing between regular boxes and converse boxes.

```
lexBox :: [Char] -> [Token]
lexBox cs =
    case break (==']') cs of
       ([],     h:rest) -> (TokenBox "0") : lexer rest
       (['~'],  h:rest) -> (TokenCbox "0") : lexer rest
       (rel, h:rest) -> case readRel rel of
                      Just (num,False) -> (TokenBox num)  : lexer rest
                      Just (num,True)  -> (TokenCbox num) : lexer rest
```

Recognize diamonds, while distinguishing between regular and converse.

```
lexDia :: [Char] -> [Token]
lexDia cs =
    case break (=='>') cs of
      ([],     h:rest) -> (TokenDia "0") : lexer rest
      (['~'],  h:rest) -> (TokenCdia "0") : lexer rest
      (rel,    h:rest) -> case readRel rel of
                Just (num,False) -> (TokenDia num)  : lexer rest
                Just (num,True)  -> (TokenCdia num) : lexer rest
```

Read the contents of a box or diamond expression:

```
readRel :: [Char] -> Maybe ([Char],Bool)
readRel (' ':rel) = readRel rel
readRel ('R':rel) = readRel rel
readRel      rel  =
    case span isDigit rel of
       (num, res) -> if null num then Nothing else Just (num,isConv res)
```

Check whether a relation is regular or converse:

```
isConv :: [Char] -> Bool
isConv (' ':xs) = isConv xs
isConv ('~':_)  = True
isConv  _       = False
```

Scan and tokenize names:

```
lexName cs =
   case span isAlphaDigit cs of
      ("true",rest)   -> TokenTrue      : lexer rest
      ("false",rest)  -> TokenFalse     : lexer rest
      ("T",rest)      -> TokenTrue      : lexer rest
      ("F",rest)      -> TokenFalse     : lexer rest
      ("P",rest)      -> TokenProp "0"  : lexer rest
      ('P':name,rest) -> TokenProp name : lexer rest
      ("p",rest)      -> TokenProp "0"  : lexer rest
      ('p':name,rest) -> TokenProp name : lexer rest
      ("C",rest)      -> TokenCst  "0"  : lexer rest
      ('C':name,rest) -> TokenCst  name : lexer rest
      ("c",rest)      -> TokenCst "0"   : lexer rest
      ('c':name,rest) -> TokenCst  name : lexer rest
      ("X",rest)      -> TokenVar "0"   : lexer rest
      ('X':name,rest) -> TokenVar  name : lexer rest
      ("x",rest)      -> TokenVar "0"   : lexer rest
      ('x':name,rest) -> TokenVar  name : lexer rest
    where isAlphaDigit c = isAlpha c || isDigit c
```

Appendix 2: Source for 'Happy' Parser Generator

See `http://www.haskell.org/happy/` for information about the format.

```
{
module HylotabParse

where

import HylotabLex
```

```
import Form

}

%name parse
%tokentype { Token }

%token
            at1             { TokenAt1 }
            at2             { TokenAt2 }
            prop            { TokenProp $$ }
            cst             { TokenCst $$ }
            var             { TokenVar $$ }
            true            { TokenTrue }
            false           { TokenFalse }
            neg             { TokenNeg }
            and             { TokenAnd }
            conj            { TokenConj }
            or              { TokenOr }
            disj            { TokenDisj }
            dimp            { TokenDimp }
            impl            { TokenImpl }
            a               { TokenA }
            e               { TokenE }
            box             { TokenBox $$ }
            cbox            { TokenCbox $$ }
            dia             { TokenDia $$ }
            cdia            { TokenCdia $$ }
            bnd             { TokenBnd }
            '('             { TokenOB }
            ')'             { TokenCB }
            ','             { TokenComma }
            '.'             { TokenDot }
%right impl
%right dimp
%left or
%left and
%left box cbox dia cdia neg
%%

Form :
       cst                     { Nom (C (read $1)) }
     | var                     { Nom (V (read $1)) }
     | prop                    { Prop (read $1)    }
     | true                    { Bool True }
     | false                   { Bool False }
```

```
     | a Form                    { A $2  }
     | e Form                    { E $2  }
     | dia Form                  { Dia (read $1) $2  }
     | cdia Form                 { Cdia (read $1) $2 }
     | box Form                  { Box (read $1) $2  }
     | cbox Form                 { Cbox (read $1) $2 }
     | '(' Form dimp Form ')'    { Conj [Impl $2 $4,Impl $4 $2]  }
     | '(' Form impl Form ')'    { Impl $2 $4 }
     | neg Form                  { Neg $2 }
     | '(' Form and Form ')'     { Conj (flattenConj [$2,$4])  }
     | '(' Form or Form ')'      { Disj (flattenDisj [$2,$4])  }
     | cst at1 Form              { At (C (read $1)) $3 }
     | var at1 Form              { At (V (read $1)) $3 }
     | at2 cst Form              { At (C (read $2)) $3 }
     | at2 var Form              { At (V (read $2)) $3 }
     | bnd var Form              { Down (read $2) $3 }
     | bnd var '.' Form          { Down (read $2) $4 }
     | conj '(' ')'              { Bool True }
     | conj '(' Forms ')'        { Conj (reverse $3) }
     | disj '(' ')'              { Bool False }
     | disj '(' Forms ')'        { Disj (reverse $3) }
     | '(' Form ')'              { $2 }

Forms : Form                     { [$1] }
      | Forms ',' Form           { $3 : $1 }

{
happyError :: [Token] -> a
happyError _ = error "Parse error"
}
```