Formules logiques en haskell

Fabian Reiter
`fabian.reiter@univ-eiffel.fr`

Stéphane Vialette
`stephane.vialette@univ-eiffel.fr`

17 novembre 2020

1 Travail demandé

Le travail est à faire en binôme.

Le projet est à rendre sous forme d'un dépôt git (github si possible) au plus tard le 18 décembre 2020 à 12h. Il devra pouvoir être complilé par `stack build` après clonage du dépôt et l'accès à la bibliothèque par `stack exec - ghci`.

Une attention particulière sera portée à la clareté du code. En particulier, privilégiez la composition et la réutilisation de fonctions simples. « *Premature optimization is the root of all evil* ». D. Knuth.

N'oubliez pas de documenter - succintement - vos modules et fonctions. La génération de la documentation est déclenchée par `stack haddock` (`https://www.haskell.org/haddock/doc/html/markup.html`). Des tests unitaires ne sont pas demandés mais seront favorablement considérés ☺ (`stack test`). La section 10 (page 11) propose quelques pistes pour aller plus loin si vous avez terminé.

2 Introdution

Ce projet - qui adopte plus la forme d'un TP long- est en lien avec le TP 4 précédent. Mais, alors que le TP 4 demandait de satisfaire des formules CNF, ce projet se focalise sur la manipulation et la transformation de formules (pas nécessairement CNF). En quelque sorte, c'est une brique logicielle préalable à la résolution des formules, il faut en effet pouvoir construire des formules complexes, les transformer et les simplifier avant de chercher à les résoudre.

Les formules se construisent à l'aide de *connecteurs logiques*. On distingue les connecteurs suivants :

— la *négation* (**NOT**) d'une formule P, notée $\neg P$,

— la *conjonction* (**AND**) de deux formules P et Q, notée $P \wedge Q$,
— la *négation de la conjonction* (**NAND**) de deux formules P et Q, notée $P \uparrow Q$,
— la *disjonction* (**OR**) de deux formules P et Q, notée $P \vee Q$,
— la *négation de la disjonction* (**NOR**) de deux formules P et Q, notée $P \downarrow Q$,
— la *disjonction exclusive* (**XOR**) de deux formules P et Q, notée $P \oplus Q$,
— la *négation de la disjonction exlusive* (**XNOR**) de deux formules P et Q, notée $P \odot Q$,
— l'*implication* (**IMPLY**) de deux formules P et Q, notée $P \Rightarrow Q$, et
— l'*équivalence* (**EQUIV**) de deux formules P et Q, notée $P \Leftrightarrow Q$.

Les tables de vérités sont les suivantes :

P	Q	$\neg P$	$P \wedge Q$	$P \uparrow Q$	$P \vee Q$	$P \downarrow Q$	$P \oplus Q$	$P \odot Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
0	0	1	0	1	0	1	0	1	1	1
0	1	1	0	1	1	0	1	0	1	0
1	0	0	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	1	1	1

Quelques rappels élémentaires. Les *lois de De Morgan* sont des identités entre propositions logiques. Pour deux formules P et Q, nous avons :

$$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$$
$$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q.$$

Distributivité de $\vee$ *sur* $\wedge$ et *distributivité de* $\wedge$ *sur* $\vee$. Pour trois formules P, Q et R, nous avons :

$$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$$
$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R).$$

3 Les modules

Notre projet sera dans `Data.Logic`. Il sera composé au final au moins des modules suivants :

— `Data.Logic.Var` pour les variables,
— `Data.Logic.Var.Some` pour accéder rapidement à quelques variables (comme dans le TP 4),
— `Data.Logic.Fml` pour les formules,
— `Data.Logic.Fml.Some` pour accéder rapidement à quelques formules (comme dans le TP 4),
— `Data.Logic.Fml.Combinator` pour la boite à outils.

Vous pouvez bien sûr ajouter d'autres modules à votre projet. Les modules `Data.Logic.Var.Some` et `Data.Logic.Fml.Some` ne sont pas demandés explicitement dans l'énoncé mais vous devez construire dans ces modules des variables et des formules pour tester toutes vos fonctions.

4 Les types haskell

Pour les variables, nous utiliserons obligatoirement le type suivant :

```haskell
newtype Var a = Var { getName :: a } deriving (Eq, Ord)
```

Le module `Data.Logic.Var` est donné complet en 11 (page 12). Vous remarquerez que c'est exactement le module que nous avons utilisé dans le TP 4.

Pour les formules, nous utiliserons obligatoirement le type suivant :

```haskell
import Data.Logic.Var as Var

data Fml a = And   (Fml a) (Fml a)
           | NAnd  (Fml a) (Fml a)
           | Or    (Fml a) (Fml a)
           | NOr   (Fml a) (Fml a)
           | XOr   (Fml a) (Fml a)
           | XNOr  (Fml a) (Fml a)
           | Imply (Fml a) (Fml a)
           | Equiv (Fml a) (Fml a)
           | Not   (Fml a)
           | Final (Var.Var a)
           deriving (Show)
```

Le module `Data.Logic.Fml` est donné incomplet en 12 (page 13).

Vous remarquerez que, contrairement au TP 4, nous n'avons pas un type spécifique pour les littéraux (c'est une différence importante). Le littéral (positif) x est représenté par `Fml.Final (Var.mk "x")` [1] alors que le littéral (négatif) $\neg x$ est représenté par `Fml.Not (Fml.Final (Var.mk "x"))` [2]. Autrement dit, un littéral est une formule (`Data.Logic.Fml`).

Quelques formules très simples :

```
>>> vx = Final $ Var.mk "x"
>>> vy = Final $ Var.mk "y"
>>> vz = Final $ Var.mk "z"
>>> vx
Final "x"
```

1. C'est-à-dire la fonction `Fml.Final . Var.mk`.
2. C'est-à-dire la fonction `Fml.Not . Fml.Final . Var.mk`.

```
>>> Not vx
Not (Final "x")
>>> And vx vy
And (Final "x") (Final "y")
>>> Or vy (Not vz)
Or (Final "y") (Not (Final "z"))
>>> Equiv (And vx vy) (Or vy (Not vz))
Equiv (And (Final "x") (Final "y")) (Or (Final "y") (Not (Final "z")))
```

et en utilisant `prettyFormat`

```
>>> putStrLn $ prettyFormat vx
"x"
>>> putStrLn . prettyFormat $ Not vx
-"x"
>>> putStrLn . prettyFormat $ And vx vy
("x" . "y")
>>> putStrLn . prettyFormat $ Or vy (Not vz)
("y" + -"z")
>>> putStrLn . prettyFormat $ Equiv (And vx vy) (Or vy (Not vz))
(("x" . "y") <=> ("y" + -"z"))
```

5 Requêtes

En guise d'échauffement.

5.1 Variables

Écrire la fonction

```
-- |'vars' @p@ returns all variables that occur in formula @p@. Duplicate
--  occurrences are removed.
vars :: (Eq a) => Fml a -> [Var.Var a]
```

du module `Data.Logic.Fml` qui calcule la liste des variables distinctes qui apparaisent dans une formules donnée.

5.2 Profondeur

Écrire la fonction

```
-- |'depth' @p@ returns the depth of fomula @p@.
depth :: (Num b, Ord b) => Fml a -> b
```

du module `Data.Logic.Fml` qui calcule la *profondeur* d'une formule. La profondeur d'une formule est définie comme le nombre maximal de fonctions logiques menant à une variable. Pour toute variable x, la profondeur de la formule (x)

est égale à 0 et celle de la formule $(\neg x)$ est égale à 1. Pour toutes formules P et Q et tout connecteur logique $\circ$, la profondeur de $(P \circ Q)$ est égale à un plus le maximum des profondeurs des formules P et Q.

```
>>> depth . Final $ Var.mk "x1"
0
>>> depth . Not . Final $ Var.mk "x1"
1
>>> depth . Not . Not . Final $ Var.mk "x1"
2
>>> depth $ Or (Final $ Var.mk "x1") (Not . Final $ Var.mk "x2")
2
```

6 Transformations

6.1 Negation Normal Form (NNF)

Une formule est NNF (*Negation Normal Form*) si l'opérateur de la négation ($\neg$) est appliqué uniquement aux variables, et les seuls opérateurs booléens autorisés sont la conjonction ($\wedge$) et la disjonction ($\vee$). Attention, la forme normale négative n'est pas une forme canonique, par exemple, $x_1 \wedge (x_2 \vee \neg x_3)$ et $(x_1 \wedge x_2) \vee (x_1 \wedge \neg x_3)$ sont équivalentes, et sont toutes deux NNF.

La procédure pour transformer une formule en une formule équivalente NNF est la suivante :

— éliminer les connecteurs $\uparrow$, $\downarrow$, $\oplus$, $\odot$, $\Rightarrow$ et $\Leftrightarrow$[3],
— utiliser les lois de De Morgan pour distribuer $\neg$ sur $\wedge$ et $\vee$[4], et
— supprimer les double négations (*i.e.* $\neg\neg F \Leftrightarrow F$).

Écrire la fonction

```
-- |'toNNF' @f@ converts the formula @f@ to NNF.
toNNF :: Fml a -> Fml a
```

du module `Data.Logic.Fml` qui transforme une formule donnée en une formule équivalente NNF.

6.2 Forme normale conjonctive(CNF)

Une formule est CNF (*Conjunctive Normal Form*) si elle se compose d'une conjonction de clauses, où une clause est une disjonction de littéraux. Par exemple, $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3)$ est CNF. Par contre, $\neg(x_1 \wedge x_2)$ et $x_1 \wedge (x_2 \vee (x_3 \wedge x_4))$ ne sont pas CNF.

3. Utilisez les tables de vérité pour réécrire ces connecteurs à l'aide de $\neg$, $\wedge$ et $\vee$.
4. Il s'agit donc de « *pousser* » récursivement les négations pour s'assurer qu'elles ne sont appliquées qu'aux variables.

C'est le format des formules du TP 4 (le type haskell est néanmoins différent). Dans la partie optionnelle « *Pour aller plus loin* » page 11 il vous est demandé de faire le point entre les deux types).

La procédure pour transformer une formule en une formule équivalente CNF est la suivante :

— transformer P en une formule NNF équivalente, et
— distribuer $\vee$ sur $\wedge$.

Écrire la fonction

```
-- |'toCNF' @f@ converts the formula @f@ to CNF.
toCNF :: Fml a -> Fml a
```

du module `Data.Logic.Fml` qui transforme une formule donnée en une formule CNF équivalente.

6.3 Forme normale disjonctive (DNF)

Une formule est DNF (*Disjunctive Normal Form*) si elle se compose d'une disjonction de conjonctions. Par exemple, $(x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3 \wedge x_4) \vee (\neg x_1 \wedge \neg x_3)$ est DNF. Par contre, $\neg(x_1 \wedge x_2)$ et $x_1 \wedge (x_2 \vee (x_3 \wedge x_4))$ ne sont pas DNF.

La procédure pour transformer une formule en une formule équivalente DNF est la suivante :

— transformer P en une formule NNF équivalente, et
— distribuer $\wedge$ sur $\vee$.

Écrire la fonction

```
-- |'toDNF' @f@ converts the formula @f@ to DNF.
toDNF :: Fml a -> Fml a
```

du module `Data.Logic.Fml` qui transforme une formule donnée en une formule DNF équivalente.

6.4 Tester

Écrire les fonctions

```
-- |'isNNF' @f@ returns true iff formula @f@ is NNF.
isNNF :: Fml a -> Bool

-- |'isCNF' @f@ returns true iff formula @f@ is CNF.
isCNF :: Fml a -> Bool

-- |'isDNF' @f@ returns true iff formula @f@ is DNF.
isDNF :: Fml a -> Bool
```

du module `Data.Logic.Fml` qui testent si une formule est NNF, CNF ou DNF.

7 Universalité

Les fonctions **NAND** et **NOR** sont dites « *universelles* » car elles permettent de reconstituer toutes les autres fonctions logiques. C'est une propriété très importante car, le circuit électronique CMOS de la fonction **NAND** étant des plus simples, la fonction **NAND** sert souvent de « *brique de base* » à des circuits intégrés beaucoup plus complexes. Les règles de réécriture sont simples (il est facile de les retrouver).

$$\begin{aligned} \neg P &= P \uparrow P \\ P \vee Q &= (P \uparrow P) \uparrow (Q \uparrow Q) \\ P \wedge Q &= ((P \uparrow Q) \uparrow (P \uparrow Q)). \end{aligned}$$

et

$$\begin{aligned} \neg P &= P \downarrow P \\ P \vee Q &= (P \downarrow Q) \downarrow (P \downarrow Q) \\ P \wedge Q &= (P \downarrow P) \downarrow (Q \downarrow Q). \end{aligned}$$

Écrire les fonctions

```
-- |'toUniversalNAnd' @p@ returns a NAND-formula that is equivalent
-- to formula @p@.
toUniversalNAnd :: Fml a -> Fml a

-- |'toUniversalNOr' @p@ returns a NOR-formula that is equivalent
-- to formula @p@.
toUniversalNOr :: Fml a -> Fml a
```

du module `Data.Logic.Fml`. La fonction `toUniversalNAnd` (resp. `toUniversalNOr`) transforme une formule donnée en une formule équivalente qui n'utilise que des variables et des fonctions **NAND** (resp. **NOR**).

Écrire maintenant les fonctions

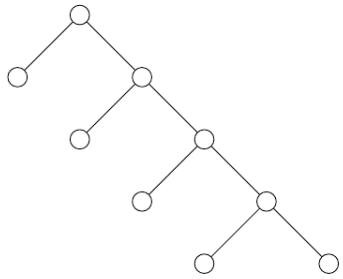
```
-- |'isUniversalNAnd' @p@ returns true iff formula @p@ uses only NAND
-- and variables.
isUniversalNAnd :: Fml a -> Bool

-- |'isUniversalNOr' @p@ returns true iff formula @p@ uses only NOR
-- and variables.
isUniversalNOr :: Fml a -> Bool
```

du module `Data.Logic.Fml`.

8 Comb Conjunctive Normal Form (CCNF)

Une formule P est CCNF (*Comb Conjunctive Normal Form*) si :
— P est une clause, ou bien
— $P = Q \wedge R$, Q est une clause et R est CCNF (une définition récursive donc).

Une formule est donc CCNF si elle est CNF et l'arbre de la formule dessine un peigne orienté à droite (les feuilles sont des clauses et les sommets internes sont des conjonctions dans le dessin ci-dessous).



Par exemple, les 3 formules $P = (x_1 \vee x_2)$, $Q = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ et $R = (x_1 \vee x_2) \wedge ((\neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3))$ sont toutes CCNF. Par contre la formule $P' = ((x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)) \wedge (x_1 \vee x_2 \vee \neg x_3)$ n'est pas CCNF (bien qu'elle soit CNF).

Écrire la fonction

```
-- |'toCCNF' @f@ converts the formula @f@ to CCNF.
toCCNF :: Fml a -> Fml a
```

du module `Data.Logic.Fml` qui transforme une formule en une formule équivalente CCNF. Écrire maintenant la fonction

```
-- |'isCCNF' @f@ returns true iff formula @f@ is CCNF.
isCCNF :: Fml a -> Bool
```

du module `Data.Logic.Fml` qui teste si une fonction est CCNF.

9 Boite à outils

Il s'agit maintenant de proposer des fonctions pour faciliter l'écriture des formules complexes. Écrire les fonctions suivantes du module `Data.Logic.Fml.Combinator` :

```
-- |'multOr' @fs@ returns the disjunction of the formulas in @fs.
-- It returns @Nothing@ if @fs@ is the empty list.
```

```haskell
--
-- >>> Combinator.multOr []
-- Nothing
-- >>> multOr [Fml.Final (Var.mk i) | i <- [1..4]]
-- Just (Or (Final 1) (Or (Final 2) (Or (Final 3) (Final 4))))
multOr :: [Fml.Fml a] -> Maybe (Fml.Fml a)

-- |'multAnd' @fs@ returns the conjunction of the formulas in @fs.
-- It returns @Nothing@ if @fs@ is the empty list.
--
-- >>> Combinator.multAnd []
-- Nothing
-- multAnd [Fml.Final (Var.mk i) | i <- [1..4]]
-- Just (And (Final 1) (And (Final 2) (And (Final 3) (Final 4))))
multAnd :: [Fml.Fml a] -> Maybe (Fml.Fml a)
```

Écrire maintenant les fonctions suivantes du module `Data.Logic.Fml.Combinator` :

```haskell
-- |'allOf' @vs@ returns a formula that is satisfiable iff all variables
-- in @vs@ are true. The function returns @Nothing@ if @vs@ is the empty list.
allOf :: [Var.Var a] -> Maybe (Fml.Fml a)

-- |'noneOf' @vs@ returns a formula that is satisfiable iff no variable
-- in @vs@ is true. The function returns @Nothing@ if @vs@ is the empty list.
noneOf :: [Var.Var a] -> Maybe (Fml.Fml a)

-- |'atLeast' @vs@ @k@ returns a formula that is satisfied iff at least @k@
-- variables in @vs@ are true. The function returns @Nothing@ if @vs@ is the
-- empty list or @k@ is non-positive or @k@ is larger than the number of
-- variables in @vs@.
atLeast :: [Var.Var a] -> Int -> Maybe (Fml.Fml a)

-- |'atLeastOne' @vs@ returns a formula that is satisfiable iff at least one
-- variable in @vs@ is true. The function returns @Nothing@ if @vs@ is the
-- empty list.
atLeastOne :: [Var.Var a] -> Maybe (Fml.Fml a)

-- |'atMost' @vs@ @k@ returns a formula that is satisfiable iff at most @k@
-- variables in @vs@ are true. The function returns @Nothing@ if @vs@ is the
-- empty list or @k@ is non-positive or @k@ is larger than the number of
-- variables in @vs@.
atMost :: [Var.Var a] -> Int -> Maybe (Fml.Fml a)

-- |'atMostOne' @vs@ returns a formula that is satisfiable iff at most one
-- variable in @vs@ is true. The function returns @Nothing@ if @vs@ is the
-- empty list.
```

```
atMostOne :: [Var.Var a] -> Maybe (Fml.Fml a)

-- |'exactly' @vs@ @k@ returns a formula that is satisfiable iff exactly @k@
-- variables in @vs@ are true. The function returns @Nothing@ if @vs@ is the
-- empty list or @k@ is non-positive or @k@ is larger than the number of
-- variables in @vs@.
exactly :: [Var.Var a] -> Int -> Maybe (Fml.Fml a)

-- |'exactlyOne' @vs@ returns a formula that is satisfiable iff exactly one
-- variable in @vs@ is true. The function returns @Nothing@ if @vs@ is the
-- empty list.
exactlyOne :: [Var.Var a] -> Maybe (Fml.Fml a)
```

Un exemple de session `ghci`.

```
>>> import qualified Data.Logic.Var as Var
>>> import qualified Data.Logic.Fml as Fml
>>> import qualified Data.Logic.Fml.Combinator as Combinator
>>> import Data.Maybe
>>> fmap Fml.prettyFormat $ Combinator.allOf [Var.mk i | i <- [1..4]]
Just "(1 . (2 . (3 . 4)))"
>>> fmap Fml.prettyFormat $ Combinator.noneOf [Var.mk i | i <- [1..4]]
Just "(-1 . (-2 . (-3 . -4)))"
>>> fmap Fml.prettyFormat $ Combinator.atLeast [Var.mk i | i <- [1..4]] 0
Nothing
>>> fmap Fml.prettyFormat $ Combinator.atLeast [Var.mk i | i <- [1..4]] 1
Just "(1 + (2 + (3 + 4)))"
>>> fmap Fml.prettyFormat $ Combinator.atLeast [Var.mk i | i <- [1..4]] 2
Just "((1 . 2) + ((1 . 3) + ((1 . 4) + ((2 . 3) + ((2 . 4) + (3 . 4))))))"
>>> fmap Fml.prettyFormat $ Combinator.atLeastOne [Var.mk i | i <- [1..4]]
Just "(1 + (2 + (3 + 4)))"
>>> fmap Fml.prettyFormat $ Combinator.atMost [Var.mk i | i <- [1..4]] 0
Nothing
>>> fmap Fml.prettyFormat $ Combinator.atMost [Var.mk i | i <- [1..4]] 1
Just "((-1 . (-2 . -3)) + ((-1 . (-2 . -4)) + ((-1 . (-3 . -4)) +
(-2 . (-3 . -4)))))"
>>> fmap Fml.prettyFormat $ Combinator.atMost [Var.mk i | i <- [1..4]] 2
Just "((-1 . -2) + ((-1 . -3) + ((-1 . -4) + ((-2 . -3) + ((-2 . -4) +
(-3 . -4))))))"
>>> fmap Fml.prettyFormat $ Combinator.atMostOne [Var.mk i | i <- [1..4]]
Just "((-1 . (-2 . -3)) + ((-1 . (-2 . -4)) + ((-1 . (-3 . -4)) +
(-2 . (-3 . -4)))))"
>>> fmap Fml.prettyFormat $ Combinator.exactly [Var.mk i | i <- [1..4]] 0
Nothing
>>> fmap Fml.prettyFormat $ Combinator.exactly [Var.mk i | i <- [1..4]] 1
Just "((1 + (2 + (3 + 4))) . ((-1 . (-2 . -3)) + ((-1 . (-2 . -4)) +
```

```
((-1 . (-3 . -4)) + (-2 . (-3 . -4))))))"
>>> fmap Fml.prettyFormat $ Combinator.exactly [Var.mk i | i <- [1..4]] 2
Just "(((1 . 2) + ((1 . 3) + ((1 . 4) + ((2 . 3) + ((2 . 4) + (3 . 4)))))) .
((-1 . -2) + ((-1 . -3) + ((-1 . -4) + ((-2 . -3) + ((-2 . -4) +
(-3 . -4)))))))"
>>> fmap Fml.prettyFormat $ Combinator.exactlyOne [Var.mk i | i <- [1..4]]
Just "((1 + (2 + (3 + 4))) . ((-1 . (-2 . -3)) + ((-1 . (-2 . -4)) +
((-1 . (-3 . -4)) + (-2 . (-3 . -4))))))"
```

10 Pour aller plus loin

10.1 Résolution

Vous remarquerez qu'il est facile de reboucler sur le TP 4 à partir d'une formule CCNF...

10.2 Simplification

Il est possible de simplifier des formules en utilisant quelques règles simples :

$$x \vee x \Leftrightarrow x$$
$$x \wedge x \Leftrightarrow x$$
$$\neg\neg x \Leftrightarrow x.$$

10.3 Des constantes

Il est possible d'augmenter le type `Fml` :

```haskell
data Fml a = And   (Fml a) (Fml a)
           | NAnd  (Fml a) (Fml a)
           | Or    (Fml a) (Fml a)
           | NOr   (Fml a) (Fml a)
           | XOr   (Fml a) (Fml a)
           | XNOr  (Fml a) (Fml a)
           | Imply (Fml a) (Fml a)
           | Equiv (Fml a) (Fml a)
           | Not   (Fml a)
           | C Bool --
           | Final (Var.Var a)
           deriving (Show)
```

C'est une première étape vers la réalisation de circuits logiques. En posant `C False` $= \bot$ et `C True` $= \top$, on retrouve facilement les propriétés de la logique propositionnelle.

11 Data.Logic.Var

```haskell
module Data.Logic.Var (
  -- * Type
  Var(..)

  -- * Constructing
, mk
) where

-- |'Var' type
newtype Var a = Var { getName :: a } deriving (Eq, Ord)

-- |Show instance
instance (Show a) => Show (Var a) where
  show = show . getName

-- |'mk' @n@ makes a propositional variable with name @n@.
--
-- >>> [mk i | i <- [1..4]]
-- [1,2,3,4]
-- >>> [mk ("x" ++ show i) | i <- [1..4]]
-- ["x1","x2","x3","x4"]
mk n = Var { getName = n }
```

12 Data.Logic.Fml

```
module Data.Logic.Fml (
  -- * Type
  Fml (..)

  -- * Querying
, depth
, vars

  -- * Formatting
, prettyFormat

  -- * Transforming
, toNNF
, toCNF
, toCCNF
, toDNF
, toUniversalNAnd

  -- * Testing
, isNNF
, isCNF
, isCCNF
, isDNF
) where

data Fml a = And   (Fml a) (Fml a)
           | NAnd  (Fml a) (Fml a)
           | Or    (Fml a) (Fml a)
           | NOr   (Fml a) (Fml a)
           | XOr   (Fml a) (Fml a)
           | XNOr  (Fml a) (Fml a)
           | Imply (Fml a) (Fml a)
           | Equiv (Fml a) (Fml a)
           | Not   (Fml a)
           | Final (Var.Var a)
           deriving (Show)
```

```haskell
-- |'prettyFormat' @p@ return a string representation of the formula @p@.
-- and :         .
-- nand:         ~.
-- or:           +
-- nor:          ~+
-- xor:          x+
-- xnor:         x~+
-- imply:        =>
-- equivalence: <=>
-- not:          -
prettyFormat :: (Show a) => Fml a -> String
prettyFormat (And   p q) = "(" ++ prettyFormat p ++ " . "   ++ prettyFormat q ++ ")"
prettyFormat (NAnd  p q) = "(" ++ prettyFormat p ++ " ~. "  ++ prettyFormat q ++ ")"
prettyFormat (Or    p q) = "(" ++ prettyFormat p ++ " + "   ++ prettyFormat q ++ ")"
prettyFormat (NOr   p q) = "(" ++ prettyFormat p ++ " ~+ "  ++ prettyFormat q ++ ")"
prettyFormat (XOr   p q) = "(" ++ prettyFormat p ++ " x+ "  ++ prettyFormat q ++ ")"
prettyFormat (XNOr  p q) = "(" ++ prettyFormat p ++ " x~+ " ++ prettyFormat q ++ ")"
prettyFormat (Imply p q) = "(" ++ prettyFormat p ++ " => "  ++ prettyFormat q ++ ")"
prettyFormat (Equiv p q) = "(" ++ prettyFormat p ++ " <=> " ++ prettyFormat q ++ ")"
prettyFormat (Not   p)   = "-" ++ prettyFormat p
prettyFormat (Final v)   = show v

-- to be completed...
```

13 Links

— Negation Normal Form (https://en.wikipedia.org/wiki/Negation_normal_form).
— Conjunctive Normal Form (https://en.wikipedia.org/wiki/Conjunctive_normal_form).
— Disjunctive Normal Form (https://en.wikipedia.org/wiki/Disjunctive_normal_form).
— NAND gate (https://en.wikipedia.org/wiki/NAND_gate).
— Électronique/Les familles MOS : PMOS, NOMS et CMOS (https://fr.wikibooks.org/wiki/%C3%89lectronique/Les_familles_MOS_:_PMOS,_NOMS_et_CMOS).
— Transistors et portes logiques (https://www.irif.fr/~carton/Enseignement/Architecture/Cours/Gates/).