

# Calculs calendaires en Haskell

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 7 Février 2008. Dernière mise à jour le 13 Février 2008

<http://www.bortzmeyer.org/calculs-calendaires-en-haskell.html>

---

Les auteurs de l'ouvrage de référence sur les calendriers, "*Calendrical Calculations*" (<http://www.bortzmeyer.org/calendrical-calculations.html>) recommandent à ceux qui veulent vraiment comprendre les calculs calendaires de les programmer eux-mêmes. Et ils ajoutent qu'il est préférable de ne pas regarder le code source qu'ils fournissent, pour suivre réellement l'algorithme. C'est ce que j'ai fait ici.

"*Calendrical Calculations*" (<http://www.bortzmeyer.org/calendrical-calculations.html>) décrit les algorithmes en langage mathématique, joliment mis en page, et qui se prête bien à une implémentation dans un langage fonctionnel. Les auteurs avaient choisi Common Lisp, j'ai refait une petite partie de leur travail (uniquement les calendriers arithmétiques, les astronomiques sont bien plus complexes) en Haskell. La plupart des formules mathématiques du livre se traduisent directement en Haskell, ce qui facilite leur vérification. Par exemple, la définition d'une année bissextile dans le calendrier grégorien se dit simplement :

```
gregorianLeapYear y =
  (y `mod` 4) == 0 &&
    ((y `mod` 100) /= 0 || y `mod` 400 == 0)
```

ce qui est, à quelques détails de syntaxe près, la définition donnée dans le livre.

Haskell dispose de grands entiers, `Integer`, dépassant les limites des entiers de la machine (`Int` en Haskell), ce qui est pratique pour certains calculs (les calculs astronomiques dépassent facilement les capacités d'une machine de 32 bits). On définit donc :

```
type Fixed = Integer
```

Les structures de données (par exemple, une date est composée d'une année, d'un mois et d'un jour) se fait avec les constructeurs de tuples de Haskell. Par exemple, une date du compte long maya se définit avec :

```
data MayanLongCount = MayanLongCountData {baktun::Fixed, katun::Fixed,
                                           tun::Fixed, uinal::Fixed, kin::Fixed}
    deriving (Eq, Show)
```

qui se lit ainsi : une date est composée d'un "baktun", d'un "katun", d'un "tun", d'un "uinal", et d'un "kin". Tous sont de type `Fixed`. Ce type `MayanLongCount` dérive des classes de type `Eq` et `Show` ce qui lui assure automatiquement une fonction d'égalité (utile pour les tests automatiques) et un affichage sommaire, notamment dans l'interpréteur. Si on ne veut pas de cet affichage par défaut, on omet `Show` et on crée sa propre instance. Ici, pour le calendrier grégorien, on choisit l'affichage à la syntaxe ISO 8601 (le code présenté ici est une version simplifiée ; ISO 8601 imposant une taille fixe aux champs comme le mois, qui doit tenir sur deux caractères, le code réel - voir le lien à la fin pour le télécharger - est plus complexe) :

```
data Gregorian = GregorianData {gregorian'year::Fixed, gregorian'month::Fixed,
                                gregorian'day::Fixed} deriving (Eq)

-- We do not derive Show because ours is better (ISO 8601 look)
instance Show Gregorian where
  show = \g -> show (gregorian'year g) ++ "-" ++
           show (gregorian'month g) ++ "-" ++
           show (gregorian'day g)
```

L'utilisation de la construction `let` qui permet de définir des variables, nous autorise à coller d'encre plus près aux définitions du livre. Ainsi, la fonction du calendrier révolutionnaire français modifié, `modifiedFrenchRevFromFixed` est presque entièrement composée de `let`, le corps de la fonction appelant juste un constructeur (la dernière ligne) :

```
modifiedFrenchRevFromFixed f =
  let approx = floor (fromIntegral (f - frenchRevEpoch + 2) /
                          (1460969 / 4000)) + 1 in
  let year = if f < fixedFromModifiedFrenchRev
              (FrenchRevolutionaryData approx 1 1) then
              approx - 1
            else
              approx
  in
  let month = floor (fromIntegral (f -
                                  fixedFromModifiedFrenchRev
                                  (FrenchRevolutionaryData year 1 1)) / 30) + 1 in
  let day = f - fixedFromModifiedFrenchRev
              (FrenchRevolutionaryData year month 1) + 1 in
  FrenchRevolutionaryData year month day
```

(`floor` est la fonction pré-définie en Haskell pour calculer une partie entière.)

L'un des points sur lequel Haskell est plus contraignant que Common Lisp est la différence de type entre entiers et réels, qui nous oblige à des conversions explicites avec la fonction pré-définie `fromIntegral`.

Testons maintenant ces fonctions avec l'environnement interactif `ghci`. Mettons qu'on veuille traduire le 7 novembre 1917 du calendrier grégorien en « date fixe » :



```

TestCase (assertBool "One day offset"
          (fixedFromIslamic (IslamicData 1613 11 26) -
            fixedFromIslamic (IslamicData 1613 11 25) == 1)),
TestCase (assertBool "One day offset over a month"
          (fixedFromIslamic (IslamicData 2453 8 1) -
            fixedFromIslamic (IslamicData 2453 7 30) == 1)),
TestCase (assertBool "One day offset over a year" -- 4532 is leap year
          (fixedFromIslamic (IslamicData 4533 1 1) -
            fixedFromIslamic (IslamicData 4532 12 30) == 1)),
TestCase (assertBool "Non leap year 1429"
          (fixedFromIslamic (IslamicData 1430 1 1) -
            fixedFromIslamic (IslamicData 1429 12 29) == 1))
]

```

(Le point de départ du calendrier musulman, l’*epoch*, est le départ de Mahomet pour Médine.) L’une des difficultés des tests à assertion est qu’il faut trouver des données fiables. On l’a vu, s’il existe plein de logiciels de calculs calendaires, beaucoup contiennent des bogues et il pourrait être imprudent de bâtir ses tests sur eux. J’ai surtout utilisé les tables données dans *“Calendrical calculations”*, annexe C, ce qui est insuffisant, rien ne dit que ces tables n’ont pas elles-mêmes des erreurs.

Un autre mécanisme de tests consiste à tester des **propriétés** des fonctions et non plus de données particulières. Pour vérifier ces propriétés, la bibliothèque Quickcheck (<http://www.cs.chalmers.se/~rjmh/QuickCheck/>) va les vérifier sur un grand nombre de données aléatoires. Par exemple, une propriété évidente est la possibilité d’un aller-retour avec la « date fixe » :

```

prop_mayanLongCountRoundtrip = forAll generateFixed $ \f ->
  let m = mayanLongCountFromFixed f in
  ((fixedFromMayanLongCount . mayanLongCountFromFixed) f == f &&
   (mayanLongCountFromFixed . fixedFromMayanLongCount) m == m)

```

Par défaut, les données générées par Quickcheck sont trop proches de zéro pour nos besoins (les bogues ne se manifestent souvent que pour les valeurs éloignées de l’origine du calendrier). On crée donc un **générateur** à nous, `generateFixed`, défini avec les **combinateurs** de QuickCheck :

```

generateFixed = do
  i <- frequency [ (5, choose(10000,1000000)),
                  (3, choose(-1000000,-10000)),
                  (2, choose(-1000,1000)) ]
  return i

```

Ce système a ses propres faiblesses puisque les différentes fonctions dépendent les unes des autres, empêchant ainsi de détecter certaines erreurs (le résultat sera cohérent même si la fonction est fausse).

Il vaut donc mieux ajouter d’autres propriétés. Une que j’ai trouvé utile est de vérifier que le jour suivant a incrémenté le jour ou le mois (et dans ce cas que le jour est le premier du mois). Pour le calendrier grégorien, cela s’écrit :

```
prop_gregorianNextDay = forAll generateFixed $ \f ->
  let nextFixed = f + 1 in
  let gday = gregorianFromFixed f in
  let nextGday = gregorianFromFixed nextFixed in
  (
    ( -- Same month
      (gregorian'year gday == gregorian'year nextGday) &&
      (gregorian'month gday == gregorian'month nextGday) &&
      (gregorian'day gday + 1 == gregorian'day nextGday)
    ) || ( -- Next month
      (gregorian'year gday == gregorian'year nextGday) &&
      (gregorian'month gday + 1 == gregorian'month nextGday) &&
      (gregorian'day nextGday == 1)
    ) || ( -- Next year
      (gregorian'year gday + 1 == gregorian'year nextGday) &&
      (gregorian'month nextGday == 1) &&
      (gregorian'day nextGday == 1)
    )
  )
```

Une telle propriété attrape beaucoup d'erreurs. Ainsi, l'erreur du livre, enregistrée sous le numéro 331 dans l'erratum de l'édition 2000 de *"Calendrical Calculations"*, et qui affecte le calendrier républicain en faisant sauter certains premiers Vendémiaire (on passe directement du dernier jour complémentaire au deux Vendémiaire) est vite détectée et Quickcheck nous dit pour quelle valeur une propriété est fausse (ici, le 810373, donc le jour qui devrait être le premier Vendémiaire 428) :

```
...
French revolutionary roundtrip: OK, passed 1000 tests.
French revolutionary next day: Falsifiable, after 199 tests:
810373
```

Pour réaliser des fonctions de conversion aisément, on passe par un pivot. En effet, si on a N calendriers, écrire N-au-carré fonctions de conversion serait trop lourd. On écrit donc, pour chaque calendrier deux fonctions **de** et **vers** les dates fixes. Par exemple, pour le calendrier égyptien, on a :

```
fixedFromEgyptian :: Egyptian -> Fixed
egyptianFromFixed :: Fixed -> Egyptian
```

Désormais, pour convertir de n'importe quel calendrier vers n'importe quel autre, il suffit de passer par la date fixe, qui sert de pivot (ce qu'on avait déjà fait pour la conversion grégorien -> julien plus haut). Ainsi, si je veux convertir du calendrier musulman vers le républicain, j'écris :

```
islamic2republican = modifiedFrenchRevFromFixed . fixedFromIslamic
```

("." étant l'opérateur de composition de fonctions). Cette fonction peut s'utiliser ainsi :

```
*Calendar.Utils> Calendar.FrenchRevolutionary.prettyShow (islamic2republican (IslamicData 1288 1 6))
"7 Germinal 79"
```

(Le 7 germinal 79 étant la date du rétablissement du calendrier républicain par la Commune de Paris.)

Une fois muni de toutes les fonctions de conversion, on peut les utiliser dans un programme principal qui affichera la date du jour. Pour lire l'horloge, on utilisera la fonction `System.Time.getClockTime` qui, comme elle réalise une entrée/sortie, est dans la monade `IO` :

```
getToday = do
  (TOD seconds _) <- getClockTime
  return (floor(fromIntegral seconds/86400) + systemTimeEpoch)

main = do
  today <- getToday
  let todayIslamic = islamicFromFixed today
      let todayGregorian = gregorianFromFixed today
      ...
  putStrLn "Today is... "
  putStrLn ("Fixed: " ++ (show today))
  putStrLn ("Gregorian: " ++ Calendar.Gregorian.prettyShow todayGregorian ++ " (" ++ show todayGregorian ++
  ...
```

Le résultat est montré ici, pour aujourd'hui :

```
% ./today
Today is...
Fixed: 733079
Egyptian: EgyptianData {egyptian'year = 2756, egyptian'month = 10, egyptian'day = 22}
Gregorian: 7 February 2008 (2008-2-7)
Julian: 2008-1-25
Islamic: Muharram 29, 1429 (1429-1-29)
Mayan (long count): 12.19.15.1.1
French revolutionary: 19 Pluviôse 216
```

Le même opérateur de composition de fonctions peut nous permettre de construire d'innombrables fonctions utiles. Supposons qu'on veuille une fonction pour connaître le jour suivant du calendrier grégorien. Il suffit de composer les fonctions « conversion en fixe », « incrémentation de 1 » et « conversion en grégorien » :

```
*Calendar.Gregorian> let nextDay = gregorianFromFixed . (+1) . fixedFromGregorian
*Calendar.Gregorian> :t nextDay
nextDay :: Gregorian -> Gregorian
*Calendar.Gregorian> nextDay (GregorianData 2008 2 11)
2008-2-12
*Calendar.Gregorian> nextDay (GregorianData 1999 12 31)
2000-1-1
*Calendar.Gregorian> nextDay (GregorianData 2000 2 28)
2000-2-29
*Calendar.Gregorian> nextDay (GregorianData 2000 2 29)
2000-3-1
```

Pour donner une petite idée de la mise en œuvre originale, voici la même fonction de conversion d'une « date fixe » vers le calendrier républicain en Common Lisp :

<http://www.bortzmeyer.org/calculs-calendaires-en-haskell.html>

```
(defun modified-french-from-fixed (date)
  ;; TYPE fixed-date -> french-date
  ;; French Revolutionary date (year month day) of fixed
  ;; $date$.
  (let* ((approx ; Approximate year (may be off by 1).
          (1+ (quotient (- date french-epoch -2)
                        1460969/4000)))
         (year (if (< date
                   (fixed-from-modified-french
                    (french-date approx 1 1)))
                   (1- approx)
                   approx))
         (month ; Calculate the month by division.
                (1+ (quotient
                    (- date (fixed-from-modified-french
                            (french-date year 1 1)))
                        30)))
         (day ; Calculate the day by subtraction.
              (1+ (- date
                    (fixed-from-modified-french
                     (french-date year month 1))))))
    (french-date year month day)))
```

Vous pouvez télécharger le code de ces programmes en (en ligne sur <http://www.bortzmeyer.org/files/Calendar.tar.gz>).