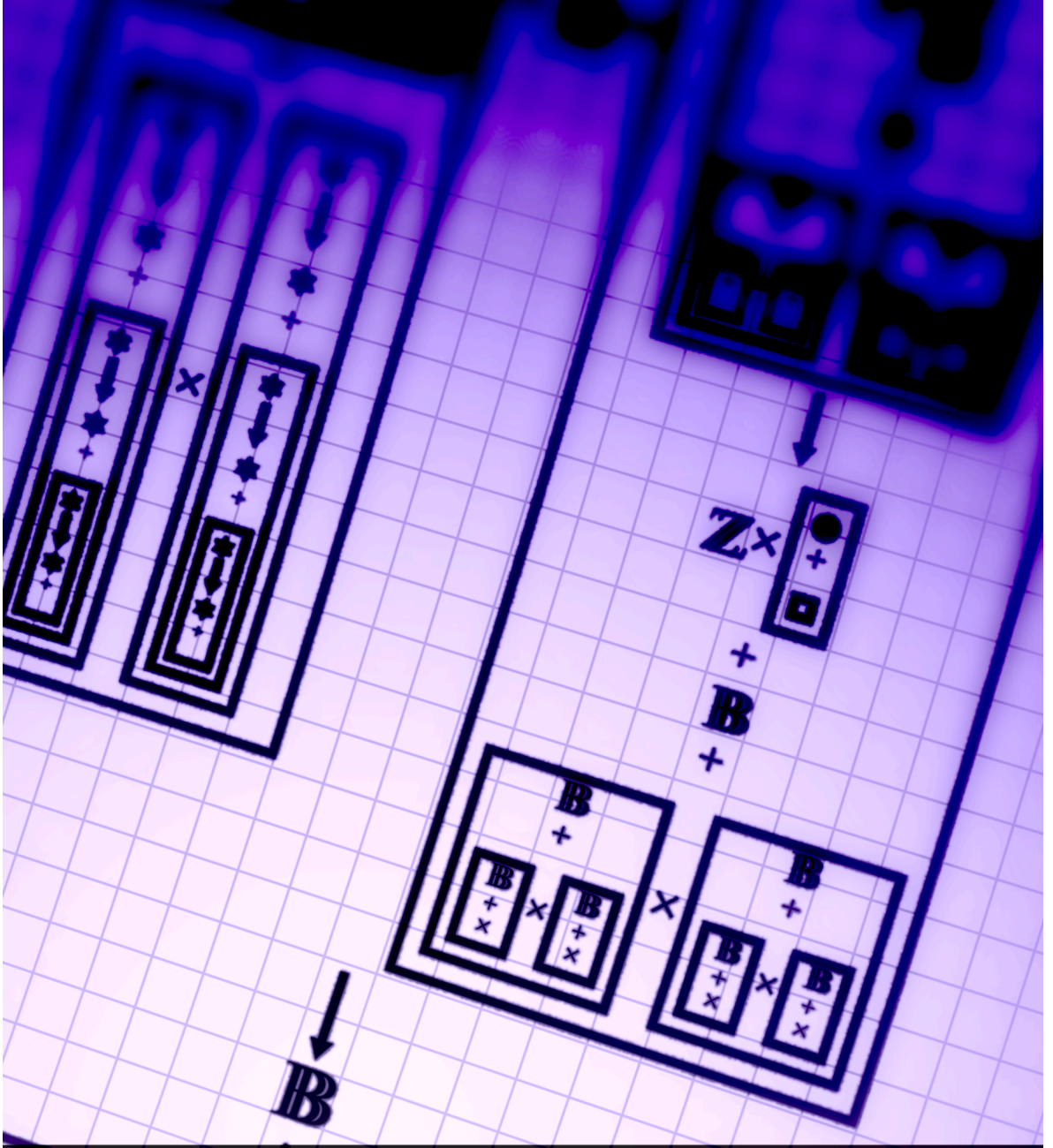


Penser en Types

Programmation au niveau Type en Haskell

Sandy Maguire

Traduit par
Jean-Hugues de Raigniac

Penser en Types

Sandy Maguire

Traduit de l'anglais par Jean-Hugues de Raigniac

Première édition

Quand les gens disent :
« Mais la plupart des bugs
de logique métier
ne sont pas des erreurs de type »,
je veux simplement leur montrer
comment faire de leurs bugs
des erreurs de type.

MATT PARSONS

Table des matières

Introduction

La programmation au niveau Type n'est pas conventionnelle. Alors que la plupart des programmeurs s'efforcent de produire plus de code qui compile, nous autres, programmeurs au niveau Type, faisons de notre mieux pour *empêcher* le code de compiler.

A proprement parler, le rôle des Types est double — ils empêchent la compilation de choses (erronées) et, ce faisant, nous guident vers des solutions plus élégantes. A titre d'exemple, si neuf des dix solutions d'un problème sont mal typées alors, trouver la bonne réponse ne réclame pas trop d'efforts.

Mais ne vous y trompez pas — le but principal de ce livre est de réduire les chances qu'un programme compile. Si vous commencez à programmer en Haskell et avez le sentiment que GHC vous pose trop souvent des problèmes, que les erreurs de typages sont trop souvent incompréhensibles, alors ce livre n'est probablement pas fait pour vous, pas encore en tout cas.

Alors à qui s'adresse ce livre? Je me suis efforcé d'écrire à ceux qui ont une maîtrise du langage intermédiaire ou avancée. Ils sont capables de résoudre de vrais problèmes en Haskell, cela sans trop batailler. Ils n'ont pas besoin d'avoir une opinion bien arrêtée sur `ExceptT` ou la levée d'exceptions dans `IO`, pas plus qu'ils ont besoin de savoir comment examiner le code Core

généré pour trouver les structures dégradant la performance.

Il est par contre attendu du lecteur visé qu'il ressente un embarras justifié au sujet des programmes qu'il écrit. Il devrait se demander, en relisant son commentaire « n'appelez pas cette fonction avec $n = 5$, ça la fait planter » s'il n'est pas possible de le faire comprendre au compilateur. De plus, même s'il est convaincu qu'ils ne seront jamais déclenchés, le lecteur devrait considérer avec angoisse les appels à `error` qu'il a dû écrire pour satisfaire le vérificateur de types.

En bref, le lecteur devrait chercher des occasions de rendre *moins* de code compilable. Ceci n'est pas motivé par des tendances masochistes, anarchistes ou du même genre. Ce désir est plutôt le fruit d'une certaine bienveillance — un peu de frustration avec le vérificateur de types maintenant est préférable à un bug difficile à trouver qui se frayera un chemin jusqu'en production.

La programmation au niveau Type, comme le reste, s'apprécie avec modération. Elle s'accompagne d'une certaine complexité et doit donc être maniée avec précaution. Alors qu'il est indispensable que votre application financière qui brasse des milliards par jour tourne sans accroc, c'est un peu moins grave si votre jeu vidéo écrit pendant vos temps libres rafraîchit mal un seul écran de jeu. Dans le premier cas, ça vaut probablement le coût de tout mettre en œuvre pour éviter que les choses tournent mal ; dans le second, cette façon de faire sera sans doute disproportionnée.

Le style est une chose réputée difficile à enseigner — de façon très concrète, le style semble être ce qui reste lorsqu'on a retiré d'une matière tout ce que nous savons expliquer. Malheureusement, savoir quand programmer au niveau Type est avant tout une question de style. Il est facile de se lâcher avec mais la retenue est céleste.

En cas de doute, prenez le parti de *ne pas* utiliser le niveau Type. Gardez ces techniques pour les cas où il serait catastrophique que ça tourne mal, lorsqu'un peu de programmation au niveau Type vous fait prendre un sérieux raccourci et lorsque cela améliore considérablement l'API. S'il n'est pas évident que vous vous retrouviez dans un de ces cas, il y a fort à parier que cela peut être fait de façon plus propre et facile au niveau Valeurs.

Mais parlons plus en détail des types en eux-mêmes.

En tant que groupe, je pense qu'on peut reconnaître que les développeurs Haskell ont l'esprit de contradiction. Je soupçonne que la majorité d'entre nous ont déjà passé une soirée à vanter les mérites d'un système fortement typé à un collègue utilisant un langage dynamique. Ils avancent des arguments du genre « j'aime Ruby parce que les types ne se mettent pas en travers de mon chemin ». Bien que, en tant que partisans de systèmes fortement typés, notre premier réflexe sera d'être consterné, je pense que c'est une critique qu'il est bon de garder à l'esprit.

En tant que développeurs Haskell, il est certain que nous avons des opinions très arrêtées sur la valeur des types. Ils *sont* utiles, ils *valent* leur pesant d'or quand nous codons, débuggons et remanions notre code. Bien sûr, nous pouvons éluder les récriminations de nos collègues d'un geste de la main et se justifier en disant qu'ils n'ont encore jamais vu un « vrai » système de types mais ce serait ne rendre un service à personne. Une réponse expéditive de ce genre passerait à côté de la raison de leur contrariété — les types se mettent *souvent* en travers de notre chemin. Nous avons juste appris à ne plus remarquer ces inconvénients plutôt que d'affronter la réalité et d'envisager l'éventualité que les types ne sont pas l'unique solution à chaque problème.

Simon Peyton Jones, l'un des auteurs principaux d'Haskell, reconnaît volontiers qu'il y a plein de programmes sans erreurs rejetés par les systèmes de types. Par exemple, examinez le programme suivant qui contient une erreur de type mais qui ne l'évalue jamais :

```
fst ("no problems", True <> 17)
```

Puisque l'erreur de type est ignorée de façon paresseuse par `fst`, l'évaluation d'une telle expression produira sans sourciller « `no problems` » à l'exécution. Bien que nous la jugions mal typée, cette expression n'en reste pas moins convenable. L'utilité d'un tel exemple est bien sûr limitée mais la démonstration est faite ; les types se mettent souvent en travers du chemin de programmes parfaitement valides.

Il arrive que cette obstruction prenne la forme de « le type que devrait avoir cette chose n'est pas clair ». On en trouve un exemple frappant dans la fonction C `printf` :

```
int printf (const char *format, ...)
```

Si vous n'avez jamais eu le plaisir d'utiliser `printf` auparavant, voilà comment ça marche : elle analyse le paramètre `format` et utilise sa structure pour dépiler les arguments supplémentaires de la pile d'appel. Comme vous le voyez, c'est la forme de `format` qui décide quels paramètres devraient figurer dans les `...` ci-dessus.

Par exemple, la chaîne de formatage `"hello %s"` prend une chaîne additionnelle et l'insère à la place de `%s`. De la même façon, le spécificateur `%d` décrit l'insertion d'un entier décimal signé.

Les appels à `printf` suivants sont tous valides :

— `printf("hello %s", "world")`, produit « hello world »,

— `printf("%d + %d = %s", 1, 2, "three")`, produit « 1 + 2 = three »,
— `printf("no specifiers")`, produit « no specifiers ».

Remarquez qu'en l'état, il semble impossible d'attribuer une signature de type haskellienne à `printf`. Les paramètres supplémentaires indiqués par son ellipse reçoivent leur type par la valeur de son premier paramètre — une chaîne. Ce genre de modèle est courant dans les langages typés dynamiquement, et dans le cas de `printf`, c'est indéniablement utile.

La documentation sur `printf` est prompte à indiquer que la chaîne de formatage ne doit pas être fournie par l'utilisateur — le faire revient à ouvrir des vulnérabilités par lesquelles un attaquant peut corrompre la mémoire et accéder au système. En fait, c'est un problème très courant — et préparer une telle chaîne est souvent le premier devoir de tout cours universitaire sur la sécurité logicielle.

Pour être clair, `printf` devient vulnérable lorsque les spécificateurs des chaînes de formatage ne correspondent pas aux arguments supplémentaires donnés. Les appels suivants à `printf`, inoffensifs en apparence, sont tous les deux malveillants.

— `printf("%d")`, corrompra probablement la pile,
— `printf("%s", 1)`, lira une quantité arbitraire de mémoire.

Le système de types du langage C n'est pas assez expressif pour décrire `printf`. Mais comme la fonction `printf` est très utile, ce n'est pas une raison suffisamment convaincante pour l'exclure du langage. Ainsi, la vérification de type est désactivée pour les appels à `printf` afin de gagner sur tous les plans. Toutefois, cela ouvre une faille par laquelle des erreurs de type peuvent s'infiltrer jusqu'à l'exécution — sous la forme de comportements imprévisibles et de problèmes de sécurité.

Selon moi, empêcher des failles de sécurité est un aspect bien plus important des types que « `null` est l'erreur à un milliard de dollar » ou tout autre argument à la mode aujourd'hui. Nous reviendrons sur le problème posé par `printf` au chapitre 9.

A de très rares exceptions, l'attitude dominante des développeurs Haskell a été d'expédier l'utilité des programmes mal typés. Comme alternative, on a une vérité inconfortable : le fait que notre langage favori n'arrive pas à faire quelque chose d'utile cependant possible avec d'autres langages.

Mais tout n'est pas perdu. En fait, Haskell *est* capable de représenter des choses aussi bizarrement typées que `printf`, pour ceux d'entre nous qui sont prêts à faire l'effort d'apprendre comment. Ce livre a pour objectif d'être *le* manuel complet pour vous conduire d'ici à là-bas, d'un programmeur Haskell compétent à celui qui persuade le compilateur de faire le travail à sa place.

Première partie

Notions de base

Chapitre 1

L'algèbre implicite des types

1.1 Isomorphismes et cardinalités

Une des fonctionnalités phares de la programmation fonctionnelle est le filtrage par motif, rendu possible par les *types de données algébriques*. Mais ce nom n'est pas juste un titre accrocheur pour des choses que l'on peut filtrer par motif. Comme leur nom le suggère, il y a une *algèbre* qui se cache derrière les types de données algébriques.

Comprendre et manipuler cette algèbre avec aisance est un superpouvoir formidable — il nous permet d'analyser les types, de leur trouver des formes plus adaptées et détermine quelles opérations (par ex. classes de types) peuvent être implémentées.

Pour commencer, nous pouvons associer à chaque type fini sa *cardinalité* — son nombre d'habitants, en ignorant les fonds. Considérez les définitions de types simples suivantes :

```haskell
data Void

data () = ()

data Bool = False | True
```

`Void` a zéro habitant, donc on lui assigne la cardinalité 0. Le type unité `()` a un habitant — donc sa cardinalité est 1. Sans vouloir insister, `Bool` a une cardinalité de 2, correspondant à ses constructeurs `True` et `False`.

Nous pouvons formuler ces déclarations sur la cardinalité ainsi :

$$
\begin{aligned}
|\texttt{Void}| &= 0 \\
|\texttt{()}| &= 1 \\
|\texttt{Bool}| &= 2
\end{aligned}
$$

Pris deux par deux, tous types finis ayant la même cardinalité seront toujours isomorphes entre eux. Un isomorphisme entre les types `s` et `t` est défini comme une paire de fonctions `to` et `from` :

```haskell
to   :: s -> t
from :: t -> s
```

de telle sorte que la composition quelconque de l'une après l'autre nous ramène là où nous avons commencé. En d'autres termes, de telle sorte que :

```
to . from = id
from . to = id
```

L'isomorphisme entre les types `s` et `t` est quelques fois écrit `s` ≅ `t`.

Si deux types ont la même cardinalité, toute association individuelle entre leurs éléments correspond exactement à ces fonctions `to` et `from`. Mais d'où vient une telle association ? D'où on veut — cela n'a pas d'importance ! Choisissez au hasard une classification pour chaque type — elle n'a pas besoin d'être une instance de `Ord` — puis associez le premier élément selon cette classification au premier élément selon l'autre. Réitérez à volonté.

Pour illustrer cela, nous pouvons définir un nouveau type ayant aussi une cardinalité de 2.

```
data Spin = Up | Down
```

Selon le raisonnement précédent, nous pouvons nous attendre à ce que `Spin` et `Bool` soient isomorphes. C'est le cas :

```
boolToSpin1 :: Bool -> Spin
boolToSpin1 False = Up
boolToSpin1 True  = Down

spinToBool1 :: Spin -> Bool
spinToBool1 Up   = False
spinToBool1 Down = True
```

Notons cependant qu'il y a un autre isomorphisme entre `Spin` et `Bool` :

```
boolToSpin2 :: Bool -> Spin
boolToSpin2 False = Down
boolToSpin2 True  = Up

spinToBool2 :: Spin -> Bool
spinToBool2 Up   = True
spinToBool2 Down = False
```

Devrions-nous préférer l'un de ces isomorphismes ? Cela a-t-il de l'importance ?

En général, pour toute paire de types de cardinalité n, il y a $n!$ isomorphismes uniques entre eux. D'un point de vue mathématique, ils se valent tous — dans la plupart des cas, il suffit de savoir qu'un isomorphisme *existe*.

Un isomorphisme entre les types `s` et `t` est une preuve, *à toutes fins utiles,* que `s` et `t` *sont la même chose.* Ils peuvent exposer des instances différentes, mais cela relève plus du fonctionnement des classes de types haskelliennes que de l'équivalence entre `s` et `t`.

Les isomorphismes sont un concept particulièrement puissant de l'algèbre de types. Nous raisonnerons en termes d'isomorphisme tout au long de ce livre, aussi est-il préférable de s'y accoutumer tout de suite.

1.2 Types Somme, Produit et Exponentiel

Dans le jargon des cardinalités, les *types somme* correspondent à l'addition. Leur exemple canonique est

`Either a b`, qui est *soit* un `a` ou un `b`. Il en résulte que la cardinalité (rappelez-vous, le nombre d'habitants) de `Either a b` est la cardinalité de `a` plus la cardinalité de `b`.

$$|\texttt{Either a b}| = |\texttt{a}| + |\texttt{b}|$$

Vous l'aurez deviné, c'est la raison pour laquelle ils s'appellent types *somme*. L'intuition derrière l'idée d'additionner s'étend à tous les types de données à constructeurs multiples — la cardinalité d'un type est toujours la somme des cardinalités de ses constructeurs.

```
data Deal a b
  = This a
  | That b
  | TheOther Bool
```

Nous pouvons analyser la cardinalité de `Deal` :

$$\begin{aligned} |\texttt{Deal a b}| &= |\texttt{a}| + |\texttt{b}| + |\texttt{Bool}| \\ &= |\texttt{a}| + |\texttt{b}| + 2 \end{aligned}$$

Nous pouvons aussi voir la cardinalité de `Maybe a`. Puisque les constructeurs de valeurs nullaires sont ennuyeux à construire — il n'y a qu'un `Nothing` — la cardinalité de `Maybe a` peut être exprimée ainsi :

$$|\texttt{Maybe a}| = 1 + |\texttt{a}|$$

Le dual des types somme est ce qu'on appelle les *types produit*. Voyons tout d'abord aussi leur exemple canonique — le type paire `(a, b)`. De façon similaire, la cardinalité d'un type produit est le *produit* des cardinalités de ses types.

$$|\texttt{(a, b)}| = |\texttt{a}| \times |\texttt{b}|$$

Pour illustrer, examinons les fractions mixtes de la forme $5\frac{1}{2}$. Nous pouvons les représenter en Haskell avec un type produit :

```haskell
data MixedFraction a = Fraction
  { mixedBit    :: Word8
  , numerator   :: a
  , denominator :: a
  }
```

Et réaliser l'analyse de sa cardinalité ainsi :

$$|\texttt{MixedFraction a}| = |\texttt{Word8}| \times |\texttt{a}| \times |\texttt{a}| = 256 \times |\texttt{a}| \times |\texttt{a}|$$

Une implication intéressante des cardinalités est la possibilité d'exprimer des *vérités mathématiques à l'aide de types*. Nous pouvons prouver par exemple que $a \times 1 = a$ en révélant un isomorphisme entre `(a, ())` et `a`.

```haskell
prodUnitTo :: a -> (a, ())
prodUnitTo a = (a, ())

prodUnitFrom :: (a, ()) -> a
prodUnitFrom (a, ()) = a
```

Dans ce cas, nous pouvons assimiler le type unité à une identité monoïdale pour les types produit — en ce sens que « l'ajouter ne change rien ». Puisque $a \times 1 = a$, nous

pouvons l'appairer avec autant de types unité que nous le désirons.

De la même façon, `Void` agit comme une unité monoïdale pour les types somme. Pour nous en convaincre, l'affirmation évidente $a + 0 = a$ peut être vue comme un isomorphisme entre `Either a Void` et `a`.

```
sumUnitTo :: Either a Void -> a
sumUnitTo (Left a)  = a
sumUnitTo (Right v) = absurd v   . . . . . .   ❶
```

```
sumUnitFrom :: a -> Either a Void
sumUnitFrom = Left
```

En ❶, la fonction `absurd` a pour type `Void -> a`. C'est une façon de bluffer en disant « si vous me donnez un `Void`, je peux vous donnez tout ce que vous voulez ». Cette promesse ne peut jamais être tenue, mais puisque nous ne pouvons jamais obtenir un `Void` en premier lieu, nous ne pouvons pas la démentir.

Les types fonction peuvent aussi être décrits en termes de cardinalité — ils correspondent à l'exponentiation. À titre d'exemple, il y a exactement quatre (2^2) habitants du type `Bool -> Bool`. Il s'agit des fonctions `id`, `not`, `const True` et `const False`. Vous pouvez vous acharner autant que vous le voulez, vous ne trouverez pas d'autres fonctions pures entre `Bool`s !

Plus généralement, le type `a -> b` a pour cardinalité $|b|^{|a|}$. Bien que cela puisse surprendre à première vue — ça me semble toujours être le contraire — le raisonnement est simple. Pour toute valeur `a` du domaine, nous voulons renvoyer un `b`. Mais nous pouvons choisir toute valeur de `b` pour chaque valeur de `a`

— ce qui nous amène à cette égalité.

$$|\texttt{a -> b}| = \underbrace{|\texttt{b}| \times |\texttt{b}| \times \cdots \times |\texttt{b}|}_{|\texttt{a}|\text{times}} = |\texttt{b}|^{|\texttt{a}|}$$

Exercice 1.2-i

Déterminez la cardinalité de `Either Bool (Bool, Maybe Bool) -> Bool`.

Le lecteur attentif pourrait se demander si la soustraction, la division ou toute autre opération mathématique ont un sens quand on les applique aux types. En fait, elles en ont un, mais c'est difficile, sinon impossible, à exprimer en Haskell. La soustraction correspond à des types auxquels on retire certaines valeurs particulières, alors que la division d'un type rend certaines de ses valeurs égales (en ce sens qu'elles sont définies de la même façon — au lieu d'avoir une instance de `Eq` qui les égaliseraient).

En fait, même la notion de calcul différentiel a un sens dans le domaine des types. Bien que nous ne développions pas davantage ce sujet, le lecteur intéressé est invité à se référer à l'article de Conor McBride « The Derivative of a Regular Type is its Type of One-Hole Contexts » [**?**].

1.3 Exemple : jeu de morpion

J'ai dit plus tôt qu'être capable de manier l'algèbre implicite des types est un superpouvoir formidable. Prouvons-le.

Imaginons que nous voulions écrire un jeu de morpion. La grille standard d'un jeu de morpion possède 9 cases qui peuvent être implémentées naïvement comme ceci :

```
data TicTacToe a = TicTacToe
  { topLeft   :: a
  , topCenter :: a
  , topRight  :: a
  , midLeft   :: a
  , midCenter :: a
  , midRight  :: a
  , botLeft   :: a
  , botCenter :: a
  , botRight  :: a
  }
```

Bien que ça marche, ce n'est pas pratique à utiliser dans un programme. Si nous voulons construire une grille vide par exemple, il y aura beaucoup de choses à remplir.

```
emptyBoard :: TicTacToe (Maybe Bool)
emptyBoard =
  TicTacToe
    Nothing Nothing Nothing
    Nothing Nothing Nothing
    Nothing Nothing Nothing
```

Ecrire des fonctions comme `checkWinner` se révèle encore plus complexe.

Plutôt que de se donner tout ce mal nous devrions utiliser notre connaissance de l'algèbre des types pour nous aider. La première étape consiste à analyser la cardinalité

de `TicTacToe` ;

$$
\begin{aligned}
|\texttt{TicTacToe a}| &= \underbrace{|\texttt{a}| \times |\texttt{a}| \times \cdots \times |\texttt{a}|}_{9 \text{ times}} \\
&= |\texttt{a}|^{9} \\
&= |\texttt{a}|^{3 \times 3}
\end{aligned}
$$

Écrites de cette façon, nous voyons que `TicTacToe` et la fonction `(Three, Three) -> a` sont isomorphes, ou dans sa forme curryfiée : `Three -> Three -> a`. `Three` est, bien sûr, n'importe quel type possédant trois habitants ; elle ressemble peut-être à ça :

```
data Three = One | Two | Three
  deriving (Eq, Ord, Enum, Bounded)
```

En raison de cet isomorphisme, nous pouvons changer la représentation de `TicTacToe` de cette manière :

```
data TicTacToe2 a = TicTacToe2
  { board :: Three -> Three -> a
  }
```

Et donc simplifier notre implémentation d'`emptyBoard` :

```
emptyBoard2 :: TicTacToe2 (Maybe Bool)
emptyBoard2 =
  TicTacToe2 $ const $ const Nothing
```

Une telle transformation ne nous permet pas de faire plus que la version précédente, mais elle améliore considérablement l'ergonomie du programme. En faisant ce changement, nous recevons en échange un jeu complet de combinateurs pour travailler avec les fonctions ; nous gagnons en compositionnalité et allégeons notre charge cognitive.

N'oublions pas que programmer est avant tout un effort humain, l'ergonomie est une quête qui en vaut la peine. Vos collègues et collaborateurs vous remercieront un jour !

1.4 L'isomorphisme de Curry-Howard

Notre discussion précédente sur les relations algébriques entre les types et leur cardinalités peut être résumé dans la table suivante.

Algèbre	**Logique**	**Types**
$a + b$	$a \vee b$	`Either a b`
$a \times b$	$a \wedge b$	`(a, b)`
b^a	$a \implies b$	`a -> b`
$a = b$	$a \iff b$	*isomorphisme*
0	$\bot$	`Void`
1	$\top$	`()`

En elle-même, cette table constitue un isomorphisme plus général entre les mathématiques et les types. Il est connu sous le nom d'*isomorphisme de Curry-Howard* — en gros, il déclare que toute expression en logique est équivalente à un programme informatique, et vice versa.

L'isomorphisme de Curry-Howard est une intuition profonde au sujet de notre univers. Il nous permet d'analyser des théorèmes mathématiques à travers le prisme de la programmation fonctionnelle. Encore mieux, il arrive souvent que des théorèmes mathématiques, même « ennuyeux », deviennent intéressants une fois exprimés sous forme de types.

Pour illustrer, considérons le théorème $a^1 = a$. Quand nous le voyons au travers de Curry-Howard, il décrit un isomorphisme entre `() -> a` et `a`. En d'autres termes, ce théorème montre qu'il n'y a pas de distinction essentielle entre avoir une valeur et avoir un programme (pur) qui calcule cette valeur. Cette perspicacité est le principe fondamental qui explique pourquoi écrire en Haskell est un tel plaisir comparé aux autres langages de programmation.

Exercice 1.4-i

Utilisez Curry-Howard pour prouver que $(a^b)^c = a^{b \times c}$. C'est-à-dire, donnez une fonction de type `(b -> c -> a) -> (b, c) -> a` et une de type `((b, c) -> a) -> b -> c -> a`. Assurez-vous qu'elles satisfassent les égalités `to . from = id` et `from . to = id`. Est-ce que ces fonctions vous rappellent quelque chose dans `Prelude` ?

Exercice 1.4-ii

Donnez une preuve de la loi des exposants selon laquelle $a^b \times a^c = a^{b+c}$.

Exercice 1.4-iii

Prouvez que $(a \times b)^c = a^c \times b^c$.

1.5 Représentations canoniques

Le fait que deux types quelconques ayant la même cardinalité sont isomorphes a pour conséquence directe la possibilité de représenter tout type de plusieurs façons. Bien que cela ne changera pas nécessairement votre façon de modeler les types, il est bon de garder à l'esprit que vous avez le choix.

Du fait de l'isomorphisme, parmi toutes les représentations d'un type, chacune est « tout aussi valable » que toutes les autres. Cependant, comme nous le verrons à la page **??**, il est souvent utile d'avoir une forme conventionnelle lorsqu'on travaille de façon générique avec les types. Cette *représentation canonique* est connue comme une *somme de produits* et se rapporte à tout type `t` de la forme,

$$t = \sum_m \prod_n t_{m,n}$$

Le gros Σ veut dire addition, et Π veut dire multiplication — nous pouvons donc lire ceci comme « addition à l'extérieur et multiplication à l'intérieur ». Nous stipulons également que toutes les additions doivent être représentées par le biais de `Either` et les multiplications par `(,)`. Ne vous inquiétez pas, écrire les règles de cette façon les rend bien plus compliquées qu'elles ne le sont en réalité.

Tout cela pour dire que chacun des types suivants est dans sa représentation canonique :

- `()`
- `Either a b`
- `Either (a, b) (c, d)`
- `Either a (Either b (c, d))`
- `a -> b`
- `(a, b)`
- `(a, Int)` — nous faisons une exception à la règle pour les types numériques, cela serait trop de travail de les représenter sous forme de sommes.

Mais aucun des types suivants n'est dans sa représentation canonique ;

- `(a, Bool)`
- `(a, Either b c)`

À titre d'exemple, la représentation canonique de `Maybe a` est `Either a ()`. Je me répète, cela ne veut pas dire que vous devriez préférer utiliser `Either a ()` plutôt que `Maybe a`. Pour le moment, il suffit de savoir que les deux types sont équivalents. Nous reviendrons sur les formes canoniques au chapitre 13.

Chapitre 3

Variance

Considérez les déclarations de type suivantes. Lesquelles d'entre elles ont des instances de `Functor` viables ?

```
newtype T1 a = T1 (Int -> a)

newtype T2 a = T2 (a -> Int)

newtype T3 a = T3 (a -> a)

newtype T4 a = T4 ((Int -> a) -> Int)

newtype T5 a = T5 ((a -> Int) -> Int)
```

Exercice 3-i

Lesquels de ces types sont des `Functor`s ? Donnez des instances pour ceux qui le sont.

Malgré toutes leurs similitudes, seuls `T1` et `T5` sont des `Functor`s. Cela est dû à la variance : si nous pouvons transformer un `a` en un `b`, cela signifie-t-il que nous pouvons nécessairement transformer un `T a` en un `T b` ?

En l'occurrence, nous pouvons parfois le faire, mais cela a beaucoup à voir avec l'aspect de `T`. Selon la forme de `T` (de genre TYPE → TYPE), il existe trois choix de variance : [1]

1. *Covariant* : Toute fonction `a -> b` peut être élevée dans une fonction `T a -> T b`.
2. *Contravariant* : Toute fonction `a -> b` peut être élevée dans une fonction `T b -> T a`.
3. *Invariant* : En général, les fonctions `a -> b` ne peuvent pas être élevées dans une fonction sur `T a`.

La covariance est celle que nous connaissons le mieux — elle correspond directement aux `Functor`s. Et en fait, le type de `fmap` représente exactement ce mouvement d'« élévation » `(a -> b) -> T a -> T b`. Un type `T` est un `Functor` si et seulement s'il est covariant.

Avant d'expliquer quand un type est covariant, regardons d'abord la contravariance et l'invariance.

1. Pour être précis, la variance est une propriété d'un type par rapport à l'un de ses constructeurs de type. Puisqu'il est entendu que les fonctions de type `map` transforment le dernier paramètre de type, nous pouvons dire sans ambiguïté que « `T` est contravariant » ; c'est un raccourci pour « `T a` est contravariant par rapport à `a` ».

Les packages `contravariant[?]` et `invariant[?]`, tous deux par Ed Kmett, nous donnent accès aux classes `Contravariant` et `Invariant`. Ces classes sont à leurs sortes de variance ce que `Functor` est à covariance.

Un type contravariant vous permet de mapper une fonction *en sens inverse* à travers son constructeur de type.

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a
```

En revanche, un type invariant `T` vous permet de mapper de `a` vers `b` si et seulement si `a` et `b` sont isomorphes. Franchement, ce n'est pas une propriété intéressante — un *isomorphisme* entre `a` et `b` signifie qu'ils sont déjà identiques au départ.

```
class Invariant f where
  invmap :: (a -> b) -> (b -> a) -> f a -> f b
```

La variance d'un type `T a` par rapport à sa variable de type `a` est entièrement spécifiée selon que `a` apparaît uniquement en *position positive*, uniquement en *position négative* ou dans un mélange des deux.

Les variables de type qui apparaissent exclusivement en position positive sont covariantes. Celles qui sont exclusivement en position négative sont contravariantes. Et les variables de type qui se retrouvent dans les deux deviennent invariantes.

Mais *qu' est-ce* qu'une position positive ou négative? Rappelons que tous les types ont une représentation canonique exprimée sous forme de combinaison de `(,)`,

`Either` et `(->)`. Nous pouvons donc définir des positions positives et négatives en fonction de ces éléments fondamentaux et développer notre intuition par la suite.

Type	Position de	
	`a`	`b`
`Either a b`	+	+
`(a, b)`	+	+
`a -> b`	−	+

La conclusion est claire : le seul moyen à notre disposition pour introduire des variables de type en position négative est de les placer du côté gauche d'une flèche. Cela devrait correspondre à votre intuition, à savoir que le type d'une fonction va « à reculons » lorsqu'il est pré-composé avec une autre fonction.

Dans l'exemple suivant, la pré-composition avec `show :: Bool -> String` transforme un type `String -> [String]` en `Bool -> [String]`.

GHCi

```
> :t words
words :: String -> [String]

> :t show :: Bool -> String
show :: Bool -> String :: Bool -> String

> :t words . (show :: Bool -> String)
words . (show :: Bool -> String) :: Bool
   ↪ -> [String]
```

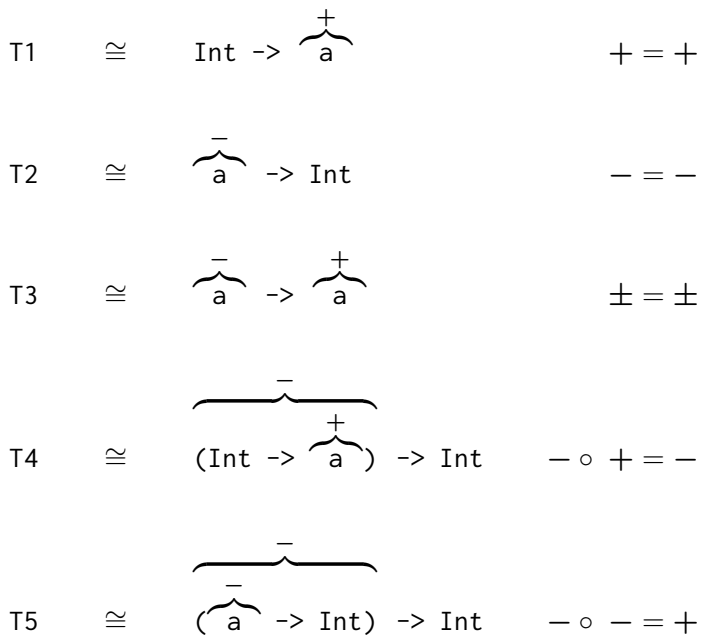
Mathématiquement, les choses sont souvent appelées « positives » et « négatives » si leurs signes suivent les lois habituelles de la multiplication. C'est-à-dire qu'un positif multiplié par un positif reste positif, un négatif multiplié par un positif est un négatif, etc.

Les variances ne sont pas différentes. Pour illustrer, considérons le type `(a, Bool) -> Int`. Le `a` dans le sous-type `(a, Bool)` est en position positive, mais `(a, Bool)` est en position négative par rapport à `(a, Bool) -> Int`. En nous souvenant de nos cours d'arithmétique élémentaire à l'école, un positif fois un négatif est négatif, et donc `(a, Bool) -> Int` est contravariant par rapport à `a`.

Cette relation peut être exprimée avec une table basique, mais encore une fois, remarquez que le nom des positions positives et négatives suggère une mnémotechnique suffisante pour garder cette table en mémoire.

a	b	a ∘ b
+	+	+
+	−	−
−	+	−
−	−	+

Nous pouvons utiliser ces connaissances pour nous convaincre que les instances de `Functor` existent uniquement pour les types `T1` et `T5` mentionnés plus haut.



Cette analyse nous montre également que `T2` et `T4` ont des instances de `Contravariant`, et `T3` a une instance d'`Invariant`.

La variance d'un type a également une interprétation plus concrète : les variables en position positive sont *produites* ou *possédées*, tandis que celles en position négative sont *consommées*. Les produits, les sommes et le côté droit d'une flèche sont tous des éléments de données qui existent déjà ou sont produits, mais le type sur le côté gauche d'une flèche est, de fait, consommé.

Il existe des noms spéciaux pour les types avec plusieurs variables de type. Un type qui est covariant dans deux arguments (comme `Either` et `(,)`) est appelé un *bifoncteur*. Un type qui est contravariant dans son premier

argument, mais covariant dans son second (comme `(->)`) est connu sous le nom de *profoncteur*. Comme vous pouvez vous en douter, Ed Kmett a des packages qui fournissent ces deux types de classes - bien que `Bifunctor` existe maintenant dans `base`.

L'analyse de position que nous venons de voir est un outil puissant — c'est rapide, vous voyez en un coup d'œil quelles instances de classe vous devez fournir. Mieux encore, c'est terriblement impressionnant pour tous ceux qui ne connaissent pas l'astuce.

Deuxième partie

Levée des restrictions

Chapitre 4

Utilisation des types

4.1 Portée du type

Haskell utilise le système de types Hindley–Milner (une généralisation). L'une des plus grandes contributions de Hindley–Milner est sa capacité à déduire les types de programmes sans avoir besoin d'annotations explicites. Le résultat est que les programmeurs Haskell au niveau du terme ont rarement besoin de prêter beaucoup d'attention aux types. Souvent, il suffit juste d'ajouter une annotation en premier. Même dans ce cas, on le fait plus pour nous que pour le compilateur.

Cet état de fait est omniprésent et le message qu'il envoie est fort et clair : « les types ne doivent pas nous ralentir dans notre réflexion ». Malheureusement, une telle attitude de la part du langage n'est pas particulièrement utile pour la programmation au niveau type. Cela se passe souvent mal — considérez la fonction suivante, qui ne compile pas *en raison* de son annotation de type :

```
broken :: (a -> b) -> a -> b
broken f a = apply
  where
    apply :: b
    apply = f a
```

Le problème avec `broken` est que, malgré les apparences, le type `b` dans `apply` n'est pas le même `b` dans `broken`. Haskell pense qu'il sait mieux que nous ici, et introduit une nouvelle variable de type pour `apply`. C'est comme si nous avions écrit à la place ce qui suit :

```
broken :: (a -> b) -> a -> b
broken f a = apply
  where
    apply :: c
    apply = f a
```

Hindley–Milner semble considérer que les types ne devraient être « ni vus ni entendus », une conséquence flagrante de ceci est que les variables de type n'ont aucune notion de portée. C'est pourquoi, l'exemple ne parvient pas à être compilé — en substance, nous avons essayé de référencer une variable non définie, et Haskell a « généreusement » pris sur lui d'en créer une nouvelle pour nous. Le Rapport Haskell ne nous fournit aucun moyen de référencer les variables de type en dehors des contextes dans lesquels elles sont déclarées.

Il existe plusieurs extensions de langage qui peuvent atténuer cette difficulté, la plus importante étant `-XScopedTypeVariables`. Lorsqu'elle est activée, elle nous permet d'assigner des variables de type et d'y faire référence plus tard. Toutefois, ce comportement est

uniquement activé pour les types qui commencent par un quantificateur explicite `forall`. Par exemple, avec `-XScopedTypeVariables`, `broken` est toujours cassé, mais ce qui suit fonctionne :

```
working :: forall a b. (a -> b) -> a -> b
working f a = apply
  where
    apply :: b
    apply = f a
```

Le quantificateur `forall a b.` introduit une portée de type et rend accessible les variables de type `a` et `b` dans le reste de la définition de la fonction. Cela nous permet de réutiliser `b` lors de l'ajout de la signature de type à `apply`, plutôt que d'introduire une *nouvelle* variable de type comme auparavant.

`-XScopedTypeVariables` nous permet de parler de types, mais nous nous retrouvons toujours sans un bon moyen d'*instancier* les types. Si nous voulions spécialiser `fmap` pour `Maybe`, par exemple, la seule solution acceptée par le Rapport Haskell est d'ajouter une signature de type intégrée.

Si nous voulions implémenter une fonction qui fournit une `String` correspondant au nom d'un type, il est difficile de savoir comment nous pourrions faire une telle chose. Par défaut, nous n'avons aucun moyen de transmettre explicitement les informations au sujet du type, et donc même *appeler* une telle fonction serait difficile.

Certaines bibliothèques plus anciennes utilisent souvent un paramètre `Proxy` pour résoudre ces problèmes. Voici sa définition :

```haskell
data Proxy a = Proxy
```

En termes de contenu d'informations au niveau valeur, `Proxy` est exactement équivalent au type unité `()`. Mais il a également un paramètre de type fantôme `a`, dont le seul but est de permettre aux utilisateurs de garder une trace d'un type et de le transmettre comme une valeur.

Par exemple, le module `Data.Typeable` fournit un mécanisme pour obtenir des informations sur les types lors de l'exécution. Il s'agit de la fonction `typeRep`, dont le type est `Typeable a => Proxy a -> TypeRep`. Encore une fois, le seul objectif de `Proxy` est de laisser `typeRep` savoir quelle représentation de type nous recherchons. De ce fait, `typeRep` doit être appelé comme ceci : `typeRep (Proxy :: Proxy Bool)`.

4.2 Applications de type

De toute évidence, le fait qu'on ne puisse pas spécifier directement les types en Haskell a de désagréables ramifications pour l'utilisateur. L'extension `-XTypeApplications` comble cette omission flagrante du langage.

Comme son nom l'indique, `-XTypeApplications` nous permet d'appliquer directement des types aux expressions. Nous pouvons remplir explicitement des variables de type en préfixant un type avec `@`. Cela peut être démontré dans GHCi :

GHCi

```
> :set -XTypeApplications

> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f
   ↪ b

> :t fmap @Maybe
fmap @Maybe :: (a -> b) -> Maybe a ->
   ↪ Maybe b
```

Alors que `fmap` élève une fonction dans n'importe quel foncteur `f`, `fmap @Maybe` élève une fonction dans `Maybe`. Nous avons appliqué le type `Maybe` à la fonction polymorphe `fmap` de la même manière que nous pouvons appliquer des valeurs aux arguments des fonctions.

Il y a deux règles à garder à l'esprit lorsque vous pensez aux applications de type. La première est que les types sont appliqués dans le même ordre qu'ils apparaissent dans une signature de type, y compris son contexte et ses quantificateurs `forall`. Cela signifie que l'application d'un type `Int` à `a -> b -> a` entraîne `Int -> b -> Int`. Mais appliquer ce type à `forall b a. a -> b -> a` conduit à `a -> Int -> a`.

Rappelons que les méthodes des classes de types ont leur contexte au début de leur signature de type. `fmap`, par exemple, a pour type `Functor f => (a -> b) -> fa -> f b`. C'est pourquoi nous avons pu remplir le paramètre foncteur de `fmap` — car il vient en premier !

La deuxième règle des applications de type est que vous pouvez éviter d'appliquer un type avec un tiret bas : `@_`. Cela signifie que nous pouvons également spécialiser

des variables de type qui ne sont pas les premières. En consultant à nouveau GHCi, nous pouvons appliquer un type aux paramètres `a` et `b` de `fmap` tout en laissant `f` polymorphe :

GHCi

```
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f
    ↪ b

> :t fmap @_ @Int @Bool
fmap @_ @Int @Bool :: Functor w => (Int
    ↪ -> Bool) -> w Int -> w Bool
```

Étant donné que les types sont appliqués dans l'ordre dans lequel ils sont définis, les types deviennent une partie de la signature publique en présence de `-XTypeApplications`. Changer l'ordre des variables de type peut casser le code en aval, soyez donc prudent lorsque vous effectuez des remaniements de cette nature.

Faites attention à l'ordre des types chaque fois que vous écrivez une fonction à laquelle des types pourraient être appliqués. Comme principe directeur, les types les plus difficiles à déduire doivent venir en premier. Cela nécessitera souvent l'utilisation de `-XScopedTypeVariables` et d'un contexte explicite pour `forall`.

`-XTypeApplications` et `-XScopedTypeVariables` sont les deux extensions les plus fondamentales de la boîte à outils d'un programmeur au niveau type. Elles vont de pair.

4.3 Types ambigus

Pour revenir à l'exemple de la fonction `typeRep` de `Data.Typeable`, nous pouvons l'utiliser pour implémenter une fonction qui nous donnera le nom d'un type. Et nous pouvons le faire sans avoir besoin du paramètre `Proxy`.

```
typeName :: forall a. Typeable a => String  ...   ❶
typeName = show . typeRep $ Proxy @a  ......   ❷
```

Il y a deux choses intéressantes à noter dans `typeName`. En ❷, `Proxy @a` est un raccourci pour `Proxy :: Proxy a` — c'est parce que le constructeur de donnée `Proxy` a pour type `Proxy t`. La variable de type `t` ici est la première de sa signature de type, nous sommes donc capables de lui appliquer un type. Les applications de type ne sont pas réservées aux fonctions, elles peuvent être utilisées partout où des types sont présents.

En ❶, nous voyons que le type `a` n'apparaît pas réellement à droite de la grosse flèche de contexte (`=>`). Étant donné que l'inférence de type de Hindley–Milner ne fonctionne qu'à droite de la flèche de contexte, cela signifie que le paramètre de type `a` dans `typeName` ne peut jamais être correctement déduit. Haskell se réfère à un tel type comme étant *ambigu*.

Par défaut, Haskell refusera de compiler tout programme contenant des types ambigus. Nous pouvons contourner ce comportement en activant l'extension bien nommée `-XAllowAmbiguousTypes` partout où nous aimerions en définir un. En fait, *utiliser* du code dont les types sont ambigus nécessitera `-XTypeApplications`.

Les deux extensions sont donc les deux faces d'une même pièce. `-XAllowAmbiguousTypes` nous permet de définir des fonctions typées de façon ambiguë, et `-XTypeApplications` nous permet de les appeler.

Nous pouvons le vérifier. En activant `-XAllowAmbiguousTypes`, nous pouvons compiler `typeName` et jouer avec.

GHCi

```
> :set -XTypeApplications

> typeName @Bool
"Bool"

> typeName @String
"[Char]"

> typeName @(Maybe [Int])
"Maybe [Int]"
```

Bien que cet exemple soit stupide, les types ambigus sont très utiles lors de la programmation au niveau type. Nous voudrons souvent mettre la main sur une représentation des types au niveau du terme — imaginez dessiner un croquis d'un type ou un programme qui exporte le schéma d'un type. Une telle fonction va presque toujours être typée de manière ambiguë, comme nous le verrons bientôt.

Cependant, les types ambigus ne sont pas toujours aussi évidents à repérer. Pour comparer, regardons un exemple surprenant. Considérez la famille de types suivante :

```
type family AlwaysUnit a where
  AlwaysUnit a = ()
```

Compte tenu de cette définition, les signatures de type suivantes sont-elles toutes non ambiguës ? Prenez un instant pour réfléchir à chaque exemple.

1. `AlwaysUnit a -> a`
2. `b -> AlwaysUnit a -> b`
3. `Show a => AlwaysUnit a -> String`

Le troisième exemple ici est, en fait, ambigu. Mais pourquoi ? Le problème est qu'il n'est pas évident de savoir quelle instance de `Show a` nous demandons ! Même s'il y a un `a` dans `Show a => AlwaysUnit a -> String`, nous ne pouvons pas y accéder — `AlwaysUnit a` est égal à `()` pour tout `a` !

Plus précisément, le problème est que `AlwaysUnit` n'a pas d'inverse ; il n'y a pas de famille de types `Inverse` telle que `Inverse (AlwaysUnit a)` est égal à `a`. En mathématiques, ce manque d'inverse est appelé *non-injectivité*.

Parce que `AlwaysUnit` est non injectif, nous ne pouvons pas apprendre ce qu'est `a`, lorsqu'on a `AlwaysUnit a`.

Prenons un exemple analogue de la cryptographie ; simplement parce que vous savez que le hachage du mot de passe de quelqu'un est `1234567890abcdef` ne signifie pas que vous savez quel est le mot de passe ; toute bonne fonction de hachage, comme `AlwaysUnit`, est *à sens unique*. Ce n'est pas parce que nous pouvons avancer que nous pouvons également revenir.

La solution à la non-injectivité consiste à donner à GHC un autre moyen de déterminer le type par ailleurs ambigu. Comme dans nos exemples, cela peut être fait en

ajoutant un paramètre `Proxy a` dont le seul objectif est de guider l'inférence, ou cela peut être accompli en activant `-XAllowAmbiguousTypes` sur le site de définition, et en utilisant `-XTypeApplications` sur le site d'appel pour remplir manuellement le paramètre ambigu.

Chapitre 5

Contraintes et TDAGs

5.1 Introduction

Les CONSTRAINTes sont étranges. Elles ne se comportent ni comme des TYPEs, ni comme les genres de donnée promue. Elles sont complètement différentes et valent donc la peine d'être étudiées.

Le genre CONSTRAINTe est réservé aux choses qui peuvent apparaître sur le côté gauche de la grosse flèche de contexte (`=>`). Cela comprend des classes de type entièrement saturées (comme `Show a`), des tuples d'autres CONSTRAINTes, et des équivalences de type (`Int ~ a`). Nous discuterons de l'équivalence de type dans un instant.

Les contraintes de classe de types sont certainement les plus familières. Nous les utilisons tout le temps, même lorsque nous n'écrivons pas au niveau type en Haskell. Considérez la fonction d'égalité `(==) :: Eq a => a -> a -> Bool`. Les tuples de CONSTRAINTes sont également bien connus : `sequenceA :: (Applicative f, Traversable t) => t (f a) -> f (t a)`

Les équivalences de type sont plus intéressantes, elles sont activées via `-XGADTs`. Comparez les deux programmes suivants :

```
five :: Int
five = 5


five_ :: (a ~ Int) => a
five_ = 5
```

`five` et `five_` sont identiques en ce qui concerne Haskell. Alors que `five` a pour type `Int`, `five_` a pour type `a`, avec une contrainte disant que `a` égal `Int`. Bien sûr, personne n'écrirait réellement `five_`, mais c'est quand même une caractéristique pratique du système de types.

Les équivalences de type forment une relation d'équivalence, ce qui signifie qu'elles ont les propriétés suivantes :

— *réflexivité* — un type est toujours égal à lui-même : `a ~ a`
— *symétrie* — `a ~ b` est valable si et seulement si `b ~ a`
— *transitivité* — si nous savons que `a ~ b` et que `b ~ c`, alors nous (ainsi que GHC) pouvons en déduire que `a ~ c`.

5.2 TDAGs

Les types de donnée algébriques généralisés (TDAGs) sont une extension du système de types Haskell qui permet d'écrire des signatures de type explicites pour les constructeurs de donnée. Comme les contraintes d'équivalence de type, ils sont également activés via

`-XGADTs`.

L'exemple de référence d'un TDAG est un arbre syntaxique sécurisé par les types. Par exemple, nous pouvons définir un petit langage avec des entiers, des booleans, l'addition, une négation logique et des déclarations de contrôle.

```
data Expr a where                    ❶
  LitInt  :: Int -> Expr Int         ❷
  LitBool :: Bool -> Expr Bool
  Add     :: Expr Int -> Expr Int -> Expr Int
  Not     :: Expr Bool -> Expr Bool
  If      :: Expr Bool -> Expr a -> Expr a -> Expr a
  ↪ ❸
```

Le `where` en ❶ est ce qui active la syntaxe TDAG pour le reste de la déclaration. Chacun des `LitInt`, `LitBool`, `Add`, etc. correspond à un constructeur de donnée du type `Expr`. Ces constructeurs prennent tous un certain nombre d'arguments avant d'aboutir à un `Expr`.

Par exemple, `LitInt` en ❷ prend un `Int` avant de renvoyer un `Expr Int`. D'autre part, le constructeur de donnée `If` en ❸ prend trois arguments (un `Expr Bool` et deux `Expr a`s) et renvoie un `Expr a`.

C'est cette capacité de spécifier le type renvoyé qui est d'un intérêt particulier.

Vous pourriez être heureux de savoir que `Expr` est *exact par construction*. Nous sommes incapables de construire un `Expr` mal typé. Bien que cela puisse ne pas sembler remarquable au premier coup d'œil, ça l'est — nous avons représenté les *règles de type d'*`Expr` dans le système de types Haskell. Par exemple, nous ne sommes pas en mesure de construire un AST qui tente d'ajouter un `Expr`

`Int` à un `Expr Bool`.

Pour nous convaincre que les signatures de type écrites en syntaxe TDAG sont en effet respectées par le compilateur, nous pouvons le vérifier dans GHCi :

GHCi

```
> :t LitInt
LitInt :: Int -> Expr Int

> :t If
If :: Expr Bool -> Expr a -> Expr a ->
   ↪ Expr a
```

Étant donné que les TDAGs nous permettent de spécifier le type d'un constructeur de donnée, nous pouvons les utiliser pour *contraindre* une variable de type dans certaines circonstances. Une telle chose n'est pas possible autrement. [1]

La valeur des TDAGs est que Haskell peut utiliser ce savoir sur ces types contraints. En fait, nous pouvons l'utiliser pour écrire un évaluateur sécurisé par les types sur `Expr` :

```
evalExpr :: Expr a -> a
evalExpr (LitInt i)  = i   . . . . . ❶
evalExpr (LitBool b) = b   . . . . . ❷
evalExpr (Add x y)   = evalExpr x + evalExpr y
evalExpr (Not x)     = not $ evalExpr x
```

1. Ou de façon similaire, comme nous le verrons, sans équivalence de type.

```
evalExpr (If b x y)  =
  if evalExpr b
     then evalExpr x
     else evalExpr y
```

En juste quelques lignes de code, nous avons obtenu un petit langage et un interprèteur pleinement fonctionnels. Voyez :

GHCi

```
> evalExpr . If (LitBool False) (LitInt
    ↪ 1) . Add (LitInt 5) $ LitInt 13
18

> evalExpr . Not $ LitBool True
False
```

Faites bien attention ici ! En ❶ , `evalExpr` renvoie un `Int`, mais en ❷ il renvoie un `Bool` ! C'est possible parce que Haskell peut *raisonner* sur les TDAGs. Dans le cas de `LitInt`, la seule possibilité pour qu'un tel motif soit sélectionné est si `a` ∼ `Int`, auquel cas on peut sans problème renvoyer un `Int`. Le raisonnement pour les autres motifs est similaire ; Haskell peut utiliser des informations depuis un motif filtré pour conduire l'inférence de type.

La syntaxe TDAG est en effet fournie par `-XGADTs`, mais ce n'est pas la syntaxe qui nous intéresse fondamentalement. L'extension est mal nommée, un nom plus approprié pourrait être « `-XTypeEqualities` ». En fait, les TDAGs ne sont que du sucre syntaxique pour les équivalences de

type. Nous pouvons également déclarer `Expr` comme un type de donnée Haskell traditionnel ainsi :

```
data Expr_ a
  = (a ~ Int)  => LitInt_  Int
  | (a ~ Bool) => LitBool_ Bool
  | (a ~ Int)  => Add_ (Expr_ Int) (Expr_ Int)
  | (a ~ Bool) => Not_ (Expr_ Bool)
  | If_ (Expr_ Bool) (Expr_ a) (Expr_ a)
```

Vu comme ça, il est un peu plus facile de comprendre ce qui se passe en coulisses. Chaque constructeur de donnée de `Expr_` porte avec lui une contrainte d'équivalence de type. Comme pour toute contrainte à l'intérieur d'un constructeur de donnée, Haskell exigera que la contrainte soit satisfaite lorsque le constructeur de donnée est appelé.

Ainsi, lorsque nous filtrons sur le motif d'un constructeur de donnée qui contient une contrainte, cette contrainte satisfaite est *propagée dans la portée*. Cela veut dire qu'une fonction de type `Expr a -> a` peut renvoyer un `Int` lorsqu'on filtre sur le motif correspondant `LitInt`, mais renvoyer un `Bool` lors d'un filtre sur `LitBool`. La contrainte d'équivalence de type `a` n'est propagée dans la portée qu'après le filtrage sur le motif du constructeur de donnée qui le contient.

Nous explorerons la technique d'encapsulation des contraintes dans les constructeurs de donnée de façon plus générale par la suite.

Bien que la syntaxe TDAG n'offre rien de nouveau, nous l'utiliserons souvent pour définir des types compliqués. C'est simplement une question de style plus lisible selon moi.

5.3 Listes hétérogènes

Extensions nécessaires

```
{-# LANGUAGE ConstraintKinds      #-}
{-# LANGUAGE DataKinds            #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE GADTs                #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE TypeFamilies         #-}
{-# LANGUAGE TypeOperators        #-}
{-# LANGUAGE UndecidableInstances #-}
```

Insertions nécessaires

```
import Data.Kind (Constraint, Type)
```

L'une des principales motivations des TDAGs est la construction de structures inductives de niveau type à partir de données de niveau terme. Comme exemple de travail pour cette section, nous pouvons utiliser des TDAGs pour définir une liste hétérogène — une liste qui peut contenir des valeurs de différents types.

Pour avoir une idée de ce que nous allons construire :

GHCi

```
> :t HNil
HNil :: HList '[]

> :t True :# HNil
True :# HNil :: HList '[Bool]
```

```
> let hlist = Just "hello" :# True :# HNil

> :t hlist
hlist :: HList '[Maybe [Char], Bool]

> hLength hlist
2
```

Le constructeur `HNil` ici est analogue au constructeur de liste régulière `[]`. De même, `(:#)` correspond à `(:)`. Ils sont définis comme un TDAG :

```
data HList (ts :: [Type]) where  . . . . . .  ❶
  HNil :: HList '[]  . . . . . .  ❷
  (:#) :: t -> HList ts -> HList (t ': ts)  . .  ❸
infixr 5 :#
```

En ❶ , vous remarquerez que nous avons donné au `ts` de `HList` une signature de genre explicite. Le paramètre de type `ts` est défini comme ayant un genre `[TYPE]`, parce que les types contenus y seront stockés. Bien que cette signature de genre ne soit pas strictement nécessaire — GHC saura correctement l'inférer pour nous — votre futur moi appréciera que vous l'ayez écrit. Une bonne règle de base est d'annoter *tous* les genres si l'*un* d'eux n'est pas `TYPE`.

`HList` est analogue au type familier `[]`, il doit donc définir une liste vide en ❷ appelé `HNil`, et un opérateur cons en ❸ appelé `(:#)`. [2] Ces constructeurs ont des types soigneusement choisis.

2. Les constructeurs de donnée symboliquement nommés en Haskell doivent commencer par deux points. Tout le reste est considéré comme une erreur de syntaxe par l'analyseur syntaxique.

`HNil` représente une `HList` vide. Nous pouvons le voir par le fait qu'il ne prend rien et renvoie `ts ~ '[]` — une liste vide de type.

L'autre constructeur de donnée, `(:#)`, prend deux paramètres. Son premier est de type `t`, et le second est un `HList ts`. En réponse, il renvoie un `HList (t ': ts)` — le résultat est que ce nouveau type devient le premier élément de l'autre `HList`.

Cette `HList` peut être filtrée sur son motif, tout comme nous le ferions avec des listes régulières. Par exemple, nous pouvons implémenter une fonction length :

```
hLength :: HList ts -> Int
hLength HNil      = 0
hLength (_ :# ts) = 1 + hLength ts
```

Mais avoir cette liste explicite de types à disposition nous permet de mettre en œuvre des choses beaucoup plus intéressantes. Pour l'illustrer, nous pouvons écrire une fonction `head` *totale*, quelque chose d'impossible à faire avec les listes traditionnelles.

```
hHead :: HList (t ': ts) -> t
hHead (t :# _) = t
```

Les bizarreries ne s'arrêtent pas là. Nous pouvons déconstruire n'importe quelle `HList` de longueur 3 dont le deuxième élément est un `Bool`, le montrer, et obtenir la garantie du compilateur que nous faisons une chose acceptable, bien qu'étrange.

```
showBool :: HList '[_1, Bool, _2] -> String
```

```
showBool (_ :# b :# _ :# HNil) = show b
```

Malheureusement, les mécanismes de dérivation fournis par GHC s'accomodent mal des TDAGs, ils refuseront d'écrire `Eq`, `Show` ou tout autre instance. Mais nous pouvons écrire les nôtres en fournissant un cas de base (pour `HNil`), et un cas inductif.

Le cas de base est que deux `HList`s vides sont toujours égales.

```
instance Eq (HList '[]) where
  HNil == HNil = True
```

Et inductivement, deux `HList`s consées ne sont égales que si leurs têtes et leurs queues sont égales.

```
instance (Eq t, Eq (HList ts)) => Eq (HList (t ': ts))
↪  where
  (a :# as) == (b :# bs) = a == b && as == bs
```

Exercice 5.3-i

Implémenter `Ord` pour `HList`.

Exercice 5.3-ii

Implémenter `Show` pour `HList`.

Nous avons dû écrire deux instances pour `Eq` afin de pouvoir affirmer que chaque élément de la liste avait également une instance `Eq`. Bien que cela fonctionne, c'est plutôt insatisfaisant. Alternativement, nous pouvons écrire une famille fermée de type qui enrobera `ts` dans une grande CONSTRAINTe précisant que chaque élément a un `Eq`.

```
type family AllEq (ts :: [Type]) :: Constraint where
  AllEq '[]       = ()   . . . . . . . . . . . . . . .  ❶
  AllEq (t ': ts) = (Eq t, AllEq ts)  . . . . . .  ❷
```

Comme `AllEq` est notre premier exemple d'une famille fermée de type inhabituelle, nous devrions passer un peu de temps à l'analyser. `AllEq` effectue un filtrage de motif au niveau type sur une liste de types, déterminant si oui ou non elle est vide.

Si elle est vide — ligne ❶ — nous renvoyons simplement la CONSTRAINTe unité. Notez qu'en raison de la signature de genre sur `AllEq`, Haskell interprète cela comme CONSTRAINTe plutôt que le TYPE unité.

Cependant, si `ts` est le cons de liste promu, nous construisons plutôt un tuple CONSTRAINTe en ❷. Vous remarquerez qu'`AllEq` est défini inductivement, de sorte qu'il finira par trouver une liste vide et terminera. En utilisant la commande `:kind!` dans GHCi, nous pouvons voir jusqu'où cette famille de type se développe.

GHCi

```
> :kind! AllEq '[Int, Bool]
AllEq '[Int, Bool] :: Constraint
= (Eq Int, (Eq Bool, () :: Constraint))
```

`AllEq` enrobe avec succès les [TYPE]s dans une CONSTRAINTe. Mais il n'y a rien de spécifique à `Eq` dans `AllEq` ! À la place, cela peut être généralisé en un enrobage dans n'importe quelle CONSTRAINTe `c`. Nous aurons besoin de `-XConstraintKinds` pour parler de contraintes polymorphes.

```
type family All (c :: Type -> Constraint)
                (ts :: [Type]) :: Constraint where
  All c '[]       = ()   . . . . . . . . . . . . . . .  ❶
  All c (t ': ts) = (c t, All c ts)   . . . . . .  ❷
```

Avec `All`, nous pouvons maintenant écrire notre instance `Eq` plus directement.

```
instance All Eq ts => Eq (HList ts) where
  HNil      == HNil      = True
  (a :# as) == (b :# bs) = a == b && as == bs
```

Exercice 5.3-iii

Réécrire les instances `Ord` and `Show` en termes de `All`.