Les types fantômes en OCaml

Xavier Van de Woestyne

Novembre 2014

Cet article n'a pas grand-chose d'inédit. Il s'agit d'une occasion de présenter une utilisation amusante d'un système de types au travers du langage OCaml pour répondre à une problématique pouvant être très coûteuse. Il est important de noter que **cet article utilise au moins OCaml 4**.

Avant-propos

Pour une bonne compréhension de cet article, il est requis d'avoir une connaissance sommaire du langage OCaml (connaître les types variants / disjonctions, les modules et les types abstraits et, *évidemment*, être à l'aise avec la syntaxe de OCaml). Avant de nous lancer dans le *vif* du sujet, nous commencerons par évoquer un cas pratique où les types fantômes auraient été utiles. Ensuite nous rappellerons quelques petites choses relatives à OCaml, et nous définirons *enfin* ce que sont les types fantômes. En fin d'article, quelques cas pratiques seront présentés.

Mars Climate Orbiter

Le 23 Mars 1999, la sonde "*Mars Climate Orbiter*" tente d'effectuer sa manoeuvre d'insertion autour de l'orbite de Mars via une procédure entièrement automatisée. Durant cette démarche, la radio devait impérativement être coupée durant le temps où la sonde se trouverait derrière Mars. Le lendemain, la sonde n'avait toujours pas émis de signaux radio. Elle est considérée comme perdue. La sonde avait suivi une trajectoire beaucoup trop basse (près de 140 km de dessous de ce qui était prévu) par rapport à sa vitesse et s'est donc probablement transformée en boule de feu. Une commission d'enquête révèlera vite la raison de cette erreur de trajectoire : la sonde recevait la poussée de ses micro-propulseurs en **Livres-force.seconde** (une unité de mesure anglo-saxonne) et le logiciel interne de la sonde traitait ces données comme s'il s'agissait de **Newton.seconde**. Cette non-concordance de données a entraîné des modifications de la trajectoire de la sonde, l'amenant à sa destruction et faisant perdre à la NASA près de 125 millions de dollars.

Cette erreur a des conséquences impressionnantes. Et même si l'on pourrait s'interroger sur la manière dont la NASA a pu commettre une erreur aussi grande, elle est extrêmement difficile à déceler car elle ne provoque aucune erreur de compilation, toutes les données étant traitées de manière uniforme, comme des flottants. L'enjeu de cet article est de présenter une manière élégante de prévenir ce genre d'erreur à la compilation.

Limite des variants classiques Limitons notre problème à la distinction entre des **centimètres** et des **kilomètres**, et comme fonctionnalités, des conversions :

- `cm_to_km`
- `km_to_cm`

Naïvement, lorsque j'ai été amené à lire le problème de typage soulevé par la sonde *Mars Climate Orbiter*, et de manière plus générale à la représentation d'unités de mesure, j'ai pensé à la définition d'une disjonction séparant les kilomètres des centimètres. Avec, par exemple, ceci comme implémentation :

```
exception IllegalMeasureData
type measure =
| Cm of float
| Km of float

let cm x = Cm x
let km x = Km x

let cm_to_km = function
  | Cm x -> Km (x *. 100000.0)
  | _ -> raise IllegalMeasureData

let km_to_cm = function
  | Km x -> Cm (x /. 100000.0)
  | _ -> raise IllegalMeasureData
```

Cette manière de procéder semble correcte, et si je tente une conversion sur une valeur invalide, par exemple `let test = km_to_cm (cm 10.0)`, mon code renverra une erreur `IllegalMeasureData`, et ce à la compilation. Cependant, si l'erreur se déclenche, c'est uniquement parce que la variable *test* évalue directement l'expression `km_to_cm (cm 10.0)`. Voyons ce qu'il se passe si nous essayons de compiler notre code avec cette déclaration : `let test () = km_to_cm (cm 10.0)`. Cette fois-ci, la compilation fonctionne.

Cela prouve que les erreurs ne sont pas vérifiées à la compilation. Car les fonctions `km_to_cm` et `cm_to_km` ont le type `measure -> measure`. Et donc une

incohérence telle que passer une valeur en kilomètres à la fonction `cm_to_km` ne peut être détectée réellement à la compilation.

Implémentation des types fantômes

Nous avons vu que les variants classiques ne permettent pas assez de vérification pour distinguer des données au sein d'un même type à la compilation (car oui, il serait possible de distinguer chaque unité de mesure dans des types différents, de cette manière :

```
module Measure =
struct
  type km = Km of float
  type cm = Cm of float
end
```

Cependant ce n'est absolument pas confortable et ce n'est pas réellement l'intérêt de cet article). Donc avant de définir et de proposer une implémentation, nous allons devoir (re)voir quelques outils en relation avec le langage OCaml.

Les variants polymorphiques

Bien que très utiles dans le *design* d'application, les variants possèdent des limitations. Par exemple, le fait qu'un type Somme ne puisse être enrichi de constructeurs (ce qui n'est plus tout-à-fait vrai depuis *OCaml 4.02.0*), mais aussi le fait qu'un constructeur ne puisse appartenir qu'à un seul type. Les variants polymorphes s'extraient de ces deux contraintes et peuvent même être déclarés à la volée, sans appartenir à un type prédéfini. La définition d'un constructeur polymorphe est identique à celle d'un constructeur normal (il commence par une majuscule) mais est précédée du caractère ‘.

```
# let a = `Truc 9;;
val a : [> `Truc of int ] = `Truc 9
# let b = `Truc "test";;
val b : [> `Truc of string ] = `Truc "test"
```

Comme vous pouvez le voir, je me suis servi deux fois du constructeur *‘Truc* en lui donnant des arguments à types différents et sans l'avoir déclaré.

Borne supérieure et inférieure L'usage des variants polymorphes introduit une notation de retour différente de celle des variants normaux. Par exemple :

```
let to_int = function
  | `Integer x -> x
  | `Float x -> int_of_float x;;

let to_int' = function
  | `Integer x -> x
  | `Float x -> int_of_float x
  | _ -> 0

# val to_int : [< `Float of float | `Integer of int ] -> int = <fun>
# val to_int' :[> `Float of float | `Integer of int ] -> int = <fun>
```

Ce que l'on remarque c'est que le chevron varie. Dans le cas où la fonction n'évalue *que* les constructeurs *Integer* et *Float*, le chevron est `<`. Si la fonction peut potentiellement évaluer autre chose, le chevron est `>`.

- `[< K]` indique que le type ne peut contenir que K
- `[> K]` indique que le type peut contenir au moins K

Nous verrons que cette restriction sur les entrées permettra d'affiner le typage de fonctions.

Restriction sur les variants polymorphes Les variants polymorphes ne permettent tout de même pas de faire des choses comme :

```
let truc = function
  | `A -> 0
  | `A x -> x
```

Au sein d'une *même* fonction, on ne peut pas utiliser un *même* variant avec des arguments différents. De mon point de vue, c'est plus logique que limitant. Mais rien n'empêche de faire deux fonctions qui, elles, utilisent des variants polymorphes à arguments variables.

Nommer les variants polymorphes Bien que l'on puisse les nommer à l'usage, il peut parfois être confortable de spécifier des variants polymorphes dans un type nommé (ne serait-ce que pour le confort de la réutilisation). Leur syntaxe (que nous verrons un peu plus bas) est assez proche des déclarations de variants classiques, cependant, **on ne peut pas** spécifier la borne dans la définition de type de variants polymorphes. Ce qui est parfaitement logique

car un type ouvert (donc borné) ne correspond pas à un seul type mais à une collection de types.

A la différence des variants normaux, les variants polymorphes se déclarent dans une liste dont les différentes énumérations sont séparées par un pipe. Par exemple :

```
type poly_color = [`Red of int | `Green of int | `Blue of int]
```

Il est évidemment possible d'utiliser les variants polymorphes dans la déclaration de variants normaux, par exemple :

```
type color_list =
| Empty
| Cons of ( [`Red of int | `Green of int | `Blue of int]  *  color_list)
```

Par contre, même si dans les définitions de types on ne peut pas spécifier de borne, on peut le faire dans les contraintes de types des fonctions. Et c'est grâce à cette autorisation que nous utiliserons les types fantômes avec des variants polymorphes.

Conclusion sur les variants polymorphes Les variants polymorphes permettent plus de flexibilité que les variants classiques. Cependant, ils ont aussi leurs petits soucis :

- Ils entraînent des petites pertes d'efficacité (mais ça, c'est superflu)
- Ils diminuent le nombre de vérifications statiques
- Ils introduisent des erreurs de typage très complexes

En conclusion, j'ai introduit les variants polymorphes car nous nous en servirons pour les types fantômes, cependant il est conseillé de ne les utiliser qu'en cas de réel besoin.

A l'assaut des types fantômes

Après une très longue introduction et une légère mise en place des pré-requis, nous allons expliquer ce que sont les types fantômes. Ensuite, nous évoquerons quelques cas de figure.

> Concrètement, un type fantôme n'est rien de plus qu'un type abstrait paramétré dont au moins un des paramètres n'est présent que pour donner des informations sur comment utiliser ce type.

Concrètement, voici un exemple de type fantôme : `type 'a t = float`. Si le type n'est pas abstrait, le type t sera identique à un flottant normal. Par contre, si le type est abstrait (donc que son implémentation est cachée), le compilateur le différenciera d'un type flottant.

Avec cette piètre définition on ne peut pas aller très loin. Voyons dans les sections suivantes quelques cas de figure précis d'utilisation des types fantômes.

Distinguer des unités de mesure

Si cet article a été introduit via une erreur dûe à des unités de mesure, ce n'est pas tout à fait innocent. Nous avions vu que via des variants classiques, il n'était a priori pas possible (en gardant un confort d'utilisation) de distinguer à la compilation des unités de mesures. Nous allons voir qu'au moyen des types fantômes, c'est très simple.

Par souci de lisibilité, j'utiliserai des sous-modules. Cependant, ce n'est absolument pas obligatoire.

```
module Measure :
sig
  type 'a t
  val km : float -> [`Km] t
  val cm : float -> [`Cm] t
end = struct
  type 'a t = float
  let km f = f
  let cm f = f
end
```

Ce module produit offre des fonctions qui produisent des valeurs de type `Measure.t`, mais ces données sont décorées. Les kilomètres et les centimètres ont donc une différence structurelle. Imaginons que nous enrichissions notre module d'une fonction addition, dont le prototype serait : `'a t -> 'a t -> 'a t`, et le code ne serait qu'un appel de `(+.)` (l'addition flottante) :

```
module Measure :
sig
  type 'a t
  val km : float -> [`Km] t
  val cm : float -> [`Cm] t
  val ( + ) : 'a t -> 'a t -> 'a t
end = struct
  type 'a t = float
  let km f = f
```

```
  let cm f = f
  let ( + ) =  ( +. )
end
```

Que se passe t-il si je fais l'addition de centimètres et de kilomètres (créés au moyen des fonctions `km` et `cm`) ? Le code plantera à la compilation car il est indiqué dans le prototype de la fonction que le paramètre du type t (`'a`) doit impérativement être le même pour les deux membres de l'addition. Nous avons donc une distinction, au niveau du typeur, des unités de mesure, pourtant représentées via des entiers.

Retournons à notre exemple de conversion, cette fois enrichissons notre module des fonctions de conversion :

```
module Measure :
sig
  type 'a t
  val km : float -> [`Km] t
  val cm : float -> [`Cm] t
  val ( + ) : 'a t -> 'a t -> 'a t
  val km_of_cm : [`Cm] t -> [`Km] t
  val cm_of_km : [`Km] t -> [`Cm] t
end = struct
  type 'a t = float
  let km f = f
  let cm f = f
  let ( + ) = ( +. )
  let km_of_cm f = f /. 10000.0
  let cm_of_km f = f *. 10000.0
end
```

Cette fois-ci, le typeur refusera formellement des conversions improbables. Dans cet exemple, j'aurais pu utiliser autre chose que des variants polymorphes pour distinguer mes centimètres de mes kilomètres. Cependant, nous allons voir qu'il est possible de se servir des bornes pour affiner le typeur. Et le fait de ne pas devoir les déclarer les rend agréable à utiliser.

Du HTML valide

Dans plusieurs frameworks de développement web, il arrive que l'on se serve de la syntaxe des fonctions du langage pour l'écriture de HTML. C'est le cas, par exemple de Yaws, un serveur web pour créer des applications web en Erlang, qui se sert des tuples et des atomes de erlang pour représenter du HTML. De même que Ocsigen, le framework de développement web OCaml, qui propose, entre autres, une écriture fonctionnelle. Il existe plusieurs intérêts à cet usage,

le premier étant le plus évident : c'est généralement beaucoup plus rapide à écrire !

> En effet, en HTML, beaucoup de balises doivent être fermées, les attributs doivent être entre guillemets et liés par un égal, bref, énormément de verbosité. (Une petite blague trouvée sur le site de Philip Wadler)

Dans un langage comme OCaml, le typage (et les types fantômes) nous permettront d'encoder une partie du DTD du HTML pour ne permettre de créer que des pages valides (selon le W3C) et donc amoindrir fortemment le flux de travail (ne devant plus passer par de la vérification avec les outils du W3C et sachant que si une page compile, c'est qu'elle est bonne). Un exemple classique : une balise `span` ne peut pas contenir de balise `div`. Et de manière plus générale, aucune balise de type block ne peut être contenue dans une balise de type inline.

Sous-typage, covariance et contravariance

Les variants polymorphes introduisent une notion de sous-typage dans le langage OCaml. Concrètement, un premier type (défini par des constructeurs polymorphes) fermé est un sous-type d'un autre type (défini lui aussi par des constructeurs polymorphes) fermé, si tous les constructeurs du premier sont inclus dans les constructeurs du second.

En OCaml, le fait de considérer un sous-type comme son type parent est possible au moyen d'une coersion, via l'opérateur `:>`, pour diminuer le sous-type dans son sur-type. Concrètement, la coersion d'une valeur d'un type `t1` en type `t2` s'écrit de cette manière : `valeur : t1 :> t2`. Le sous-typage des variants polymorphes introduit des règles particulières dans le cas des fonctions :

- Le type de retour d'une fonction suit la même direction de sous-typage que le type fonctionnel, on dit qu'il est **covariant** (et noté +)
- Le type du paramètre va dans le sens inverse, on dit qu'il est **contravariant** (et noté -).

Le compilateur de OCaml peut déterminer si un type est un sous-type d'un autre pour les types qu'il connaît. Il est donc impossible qu'il déduise le sous-typage des types abstraits. Pour permettre au compilateur de faire la relation de sous-typage, on annotera un type abstrait d'un + s'il est covariant et d'un - s'il est contravariant :

```
type (+'a) t (* Type covariant *)
type (-'a) t (* Type contravariant *)
```

Cette précision est importante car nous nous servirons de la covariance et de la contravariance pour produire des documents HTML statiquement typés.

Organisation par les types

Pour produire un document HTML, il faut d'abord isoler les constituants d'un document HTML. Pour ma part, j'ai décidé de fragmenter les balises en trois catégories :

- Le texte brut (pcdata)
- Les balises feuilles (`<br />`, `<hr />`) par exemple
- Les balises noeuds (des balises pouvant en contenir d'autres).

Ces balises seront elles-mêmes divisées en plusieurs sous-catégories, par exemple les balises dites *inline* (comme `<span>`), qui ne peuvent pas contenir de balises dites *block* (comme `<div>` par exemple).

Je suggère cette implémentation :

```
module HTML :
sig
  type (+'a) tag
end = struct
  type raw_tag =
    | Pcdata of string
    | Leaf of string
    | Node of string * raw_tag list
  type 'a tag = raw_tag
end
```

Le fait que les feuilles et les noeuds prennent des chaînes de caractères dans leur signature permettra de parser une arborescence HTML typée et d'en produire le HTML textuel correspondant.
Nous pouvons nous atteler à la construction de balises, au moyen de fonctions.

Construction de balises

Les balises sont assez simples à construire. Commençons par la balise `Pcdata`, dont la principale contrainte est de ne pouvoir prendre qu'une chaîne de caractères :

```
val pcdata : string -> [`Pcdata] tag
let pcdata x = Pcdata x
```

L'implémentation de `<hr />` et `<br />` est assez évidente, et ne demande pas de précisions :

```
val hr : [`Hr] tag
val br : [`Br] tag
let hr = Leaf "hr"
let br = Leaf "br"
```

Passons maintenant aux implémentations les plus intéressantes. Une `<div>` peut a priori prendre n'importe quel type de balise (ce n'est pas tout-à-fait vrai, mais nous partirons de ce principe pour l'exemple), son implémentation est donc elle aussi assez facile :

```
val div : 'a tag list -> [`Div] tag
let div children = Node ("div", children)
```

Les `<span>` sont un peu différents car ces derniers ne peuvent contenir **que** des `<span>` ou des `Pcdata`. Il faut donc restreindre leur domaine d'entrée :

```
val span : [< `Span | `Pcdata ] tag list -> [`Span] tag
let span children = Node ("span", children)
```

Cette fois, la décoration du type `tag` restreint les entrées à seulement des données `<span>` ou des `Pcdata`. (Oui, la borne est supérieure, donc on limite l'entrée à ces deux types uniquement). Pour rappel, le code général du module est :

```
module HTML :
sig
  type (+'a) tag
  val pcdata : string -> [`Pcdata] tag
  val hr : [<`Hr] tag
  val br : [<`Br] tag
  val div : 'a tag list -> [`Div] tag
  val span : [< `Span | `Pcdata ] tag list -> [`Span] tag
end = struct
  type raw_tag =
    | Pcdata of string
    | Leaf of string
    | Node of string * raw_tag list
  type 'a tag = raw_tag
  let pcdata x = Pcdata x
  let hr = Leaf "hr"
  let br = Leaf "br"
  let div children = Node ("div", children)
  let span children = Node ("span", children)
end
```

Je vous invite maintenant à saisir quelques expressions pour tester notre code, par exemple : `HTML.(span [br])` qui devrait échouer, ou encore `HTML.(span [pcdata "hello"])`, qui devrait réussir. Un autre exemple un peu plus long :

```
let open HTML in
  div [
    span [pcdata "Hello world"];
    hr;
    span [pcdata "Hello Nuki"];
    br;
  ]
```

Ce code est parfaitement valide et ne provoque donc aucune erreur. Par contre, si le dernier `<span>` avait été : `span [div [pcdata "Hello Nuki"]]`, une erreur aurait été levée.

Retour d'expérience sur la production de HTML valide

L'exercice est intéressant et permet de comprendre le rôle des variants polymorphes dans la décoration de types, via les types fantômes. C'est une méthode de ce genre (pas identique ceci dit, sans aucun doute plus riche) qui est utilisée dans TyXML, un des composants de Ocsigen pour rendre la production de XML invalide absolument impossible (il peut s'utiliser au moyen d'une extension de syntaxe).

Je trouve cette manière de procéder assez astucieuse, même si cet article n'en présente qu'une infime partie. En effet, il faudrait aller plus loin en typant les attributs, etc. Cependant, la vocation de cette section n'est que de montrer un rapide exemple d'utilisation des types fantômes.

Requêtes SQL statiquement typées

Une fois de plus, mon exemple est emprunté au cadriciel Ocsigen. En effet, dans beaucoup de langages web, lorsque l'on doit effectuer une communication avec une base de données, il est courant de construire la requête dans une chaîne de caractères, qui est un moyen expressif de construire du SQL et de l'envoyer au serveur de base de données, lui délégant la vérification de la formation de cette dernière. C'est une manière de faire peu fiable, qui provoque des erreurs d'exécution de code, ce que le programmeur OCaml aime éviter.

Macaque est une bibliothèque permettant de formuler des requêtes vérifiées à la compilation. Pour cet exemple, je ne rentrerai pas dans les détails (car c'est fort long et dépasse sans aucun doute mes compétences), mais Macaque s'appuie sur des types fantômes pour décorer des types de OCaml avec des informations relatives à SQL. Pour l'avoir utilisé, Macaque est très utile (et agréable) et offre le confort de la validation de requêtes (syntaxique et logique).

Conclusion

Je terminerai cette brève présentation en limitant la notion de type fantôme à “un (ou plusieurs labels) donnant des informations d'utilisations complémentaires à des types”. Cet étiquetage permet une vérification statique d'un bon usage des données. Les inconvénients sont l'expressivité limitée et le fait que ça introduise parfois des erreurs de types assez lourdes à lire.

Les types fantômes sont même utilisés dans la bibliothèque standard de OCaml et donc ne sont pas que des *features* un peu trop particulières.