Des unités dans le typeur

J. Garrigue & D. Ly

1: Graduate School of Mathematics, Nagoya University
464-8602 Furo-cho, Chikusa-ku, Nagoya, Japan
garrigue@math.nagoya-u.ac.jp
2: ENSTA ParisTech, 828 boulevard des Maréchaux,
91120 Palaiseau, France
dara.ly@ensta.fr

Résumé

Nous présentons dans cet article une extension du typeur d'OCaml permettant de représenter les unités de mesure. L'introduction de tels types permet un typage plus fin des valeurs numériques dans des applications telles que le calcul scientifique. Notre implémentation utilise l'algorithme de Kennedy pour l'unification des unités, mais la comparaison nécessite quant à elle un nouvel algorithme, que nous présentons en détail.

1. Introduction

Les calculs en sciences (et notamment en sciences physiques) font intervenir des quantités dimensionnées, c'est-à-dire des nombres affectés d'une unité de mesure. D'une certaine façon, ces unités sont à la physique ce que les types sont à la programmation : une façon de catégoriser les grandeurs manipulées qui permet d'une part d'en exprimer le sens, et d'autre part de se prémunir contre une certaine classe d'erreurs dans les calculs.

Les types rencontrés dans un programme sont créés à partir d'un certain nombre de types de base, que l'on compose en utilisant des constructions (tuples, enregistrements, variants, ...). De la même façon, les unités utilisées dans un champ d'étude scientifique donné sont dérivées d'un ensemble d'unités de bases par produit et quotient.

Les unités traduisent le sens physique d'une valeur numérique : ainsi exprimer une masse en kilogrammes revient à la comparer à une masse de référence, qui est un objet physique.

L'interprétation physique des unités « composées » peut être directe (par exemple m/s est interprétée comme le nombre de mètres parcourus par un objet en une seconde) ou un peu moins évidente, mais dans tous les cas une unité lie une grandeur numérique à une réalité physique : elle a une valeur sémantique.

Par ailleurs, les calculs sur des grandeurs dimensionnées doivent respecter certaines règles pour avoir un sens physique. On peut les résumer ainsi :

— l'unité du produit (respectivement quotient) de deux grandeurs est le produit (respectivement quotient) des unités de ces grandeurs
— la comparaison, l'égalité, la somme et la différence ne peuvent être faites qu'entre grandeurs de même unité.

On retrouve donc dans l'analyse dimensionnelle une démarche analogue à celle de la vérification de type : dans les deux cas il s'agit de vérifier la compatibilité de certaines valeurs avec les opérateurs qui leurs sont appliqués.

Il est par conséquent naturel de vouloir utiliser le typage pour traduire les unités physiques, et faire les vérifications associées. On se propose d'étendre le système de types d'OCaml dans cette optique.

2. État de l'art

L'idée d'introduire des types dimensionnés dans ML n'est pas nouvelle. Dès 1991, Wand et O'Keefe [8] ont expliqué comment on pouvait encoder les dimensions comme des n-uplets d'entiers, et comment l'unification de types à la ML contenant des dimensions pouvait se faire en deux temps : d'abord unification structurelle de la partie non dimensionnelle des types (types flèches et types de base), qui permet d'extraire un ensemble d'équations entre types de dimensions. Ce système d'équations peut ensuite être résolu par un simple pivot de Gauss, en exploitant la structure d'espace vectoriel de l'algèbre des dimensions.

Peu après, Kennedy [5] et Goubault [3] ont approfondi et mis cette idée en pratique dans deux variantes de Standard ML : une extension du ML Kit et HimML. Les deux ont en commun de reconnaître qu'il n'est pas nécessaire de procéder en deux temps : les équations dimensionnelles peuvent être résolues au fur et à mesure qu'elles apparaissent, via l'introduction de substitutions appropriées, exactement comme pour l'unification structurelle. Elles se distinguent cependant par le domaine utilisé pour les exposants : Goubault choisit comme Wand d'utiliser des nombres rationnels, ce qui facilite la résolution, mais Kennedy préfère se limiter à des exposants entiers, plus intuitifs, et adapte à cette fin un algorithme de résolution d'équations linéaires à coefficients entiers trouvé chez Knuth [7].

Ces deux approches furent le résultat de réflexions approfondies, cependant leurs implémentations n'ont pas eu beaucoup d'écho, pas plus que l'extension de Caml Light par Blanchet [1]. Il a fallu attendre une quinzaine d'année et que Kennedy réintroduise les type dimensionnés dans F# [6], pour qu'ils soient réellement popularisés. Plus récemment ils ont aussi été implémentés dans Haskell, sous la forme d'un greffon pour GHC [4].

Il faut aussi noter que si la plupart des travaux parlent de typage dimensionnel, ce qui sous entend un nombre fini de dimensions distinctes (longueur, temps, ...), en pratique les implémentations récentes distinguent plutôt les unités (mètres, pieds, litres, secondes, ...), que l'on distinguera même si elles sont homogènes d'un point de vue dimensionnel. En effet, si pour un physicien le plus important est de garantir l'homogénéité des dimensions, pour ne pas dire quelque chose d'absurde, pour un ingénieur éviter une erreur d'unité est tout autant important. Parmi les systèmes cités, seul celui de Goubault permet d'exprimer à la fois dimensions et unités, et d'assurer leur cohérence.

3. Présentation de l'extension

Nous commençons par une démonstration de notre extension d'OCaml [1].

```
# let d : <m> dfloat = create 17.
  and t : <s> dfloat = create 2. ;;
val d : <m> Dim.dfloat = 17.
val t : <s> Dim.dfloat = 2.
# let v = d /: t;;
val v : <m / s> Dim.dfloat = 8.5
# d +: t;;
        ^
Error: This expression has type <s> Dim.dfloat
       but an expression was expected of type <m> Dim.dfloat
       Type <s> is not compatible with type <m>
```

Sur cet exemple simple, le typeur infère l'unité du quotient $v = d/t$ et rejette l'expression $d + t$, non homogène dimensionnellement.

Nous avons ajouté au typeur d'OCaml des types permettant de représenter des unités. Dans cet exemple, la syntaxe `<m> dfloat` permet de passer au constructeur de type `dfloat` un type unité `m`. Le type paramétré `dfloat` pourra être défini de la façon suivante.

1. Elle est disponible à : `https://github.com/tournemire/ocaml/tree/dim`.

```
module Dim : sig
  type 'a dfloat = private float
  val create : float ->  <'a> dfloat
  val (+:) : <'a> dfloat -> <'a> dfloat -> <'a> dfloat
  val ( *: ) : <'a> dfloat ->  <'b> dfloat -> <'a * 'b> dfloat
  val ( /: ) : <'a> dfloat ->  <'b> dfloat -> <'a / 'b> dfloat
  val inv :  <'a> dfloat -> <1 / 'a> dfloat
  val dsqrt : <'a ^ 2> dfloat -> <'a> dfloat
end = struct
  type 'a dfloat = float
  let create f = f and ( +: ) = ( +. ) and ( *: ) = ( *. )
  let ( /: ) = ( /. )
  let inv f = 1. /. f
  let dsqrt = sqrt
end
```

Comme le montre cet exemple nous avons choisi, plutôt que de traiter un type numérique (par exemple `float`) de façon particulière, de pouvoir doter n'importe quel type d'un attribut dimensionnel, par le biais d'un type fantôme, implémenté par une abréviation privée. Cela a l'avantage d'assurer la correction de l'analyse dimensionnelle sans cacher que `dfloat` est implémenté par `float`.

Ces types peuvent représenter aussi bien des dimensions que des unités. La distinction prendrait son sens si on voulait automatiser la conversion entre unités de dimension cohérente. Nous avons estimé que se restreindre à des conversions explicites était cohérent avec la discipline de typage pratiquée en OCaml, où les coercions entre types ne génèrent pas de code. Dans notre système, chaque unité est vue comme une dimension indépendante, et est en fait un simple nom, sans définition préalable.

```
#let ft_of_m : <ft / m> dfloat = create 3.281;;
val ft_of_m : <ft / m> Dim.dfloat = 3.281
# let d' = ft_of_m *: d;;
val d' : <ft> Dim.dfloat = 55.777
```

Enfin, le dernier choix concerne l'espace des unités de mesure. Les unités sont définies par un ensemble d'unités de base composées par produit, inverse et puissance. Mais autorise-t-on les exposants d'une unité à être non entiers ? Il nous a semblé que les cas d'utilisation de telles unités étaient trop particuliers pour justifier leur implémentation, qui nécessiterait d'inclure des rationnels dans le compilateur. Nous avons donc fait le choix de nous restreindre aux exposants entiers. Il en résulte une discipline de typage plus stricte.

```
let x : <'a ^ 2> dfloat = d;;
Characters 26-27:
  let x : <'a ^ 2> dfloat = d;;
                            ^
Error: This expression has type <m> Dim.dfloat
       but an expression was expected of type <'a ^ 2> Dim.dfloat
       Type <m> is not compatible with type <'a ^ 2>
```

Si l'on définit un ensemble d'unités de base (donc indépendantes) $B = \{b_1, \cdots, b_n\}$ notre espace des unités est défini comme le module [2] sur $\mathbb{Z}$ engendré par B.

$$\mathscr{U} = \left\{ \prod_{i=1}^{n} b_i^{e_i} \; ; \; (e_1, \ldots, e_n) \in \mathbb{Z}^n \right\}$$

2. Un module est à un anneau ce qu'un espace vectoriel est à un corps.

4. Extension du langage et du système de types

Comme dans les exemples précédents, nous utilisons la grammaire suivante pour les unités :

$$u ::= 1 \mid \delta \mid b \mid u_1 * u_2 \mid u_1/u_2 \mid u_1\hat{}\, int$$

où δ désigne une variable dimensionnelle (variable de type qui ne peut être instanciée qu'en une unité), b une unité de base, et *int* un entier signé. 1 est l'unité des grandeurs sans dimensions. Les unités de base sont traitées de façon structurelle : comme pour les constructeurs de variants polymorphes, seul leur nom compte.

De façon interne, une unité est définie dans sa forme normale par :
— une liste de couples variable de type et exposant ;
— une liste de couples unité de base et exposant.
Les unités s'ajoutent comme un cas supplémentaire à `type_desc`, qui représente les types d'OCaml.

Le fait de considérer les unités comme des types à part entière permet de les passer à des constructeurs comme on passerait n'importe quel autre type. On évite de cette façon de traiter le type `float` (par exemple) de façon particulière, et on permet de dimensionner des vecteurs, des complexes... En termes d'implémentation cela permet aussi d'éviter de séparer variables de types et variables d'unités, alors que les opérations effectuées dessus afin de réaliser l'inférence sont essentiellement les mêmes : substitution, instanciation, et généralisation, pour citer les plus courantes.

Cette façon de procéder laisse la possibilité de créer des types dépourvus de sens comme `<m> list`. Nous avons estimé que cette possibilité n'était pas dérangeante.

5. Unification

OCaml utilise un système de types à la Hindley-Milner[2], où l'unification de deux types se fait par l'algorithme de Robinson. La nature algébrique des unités impose de les unifier de façon assez différente.

On peut considérer l'unification de deux unités comme une équation linéaire à résoudre en nombres entiers. Kennedy donne un algorithme pour la résolution d'une telle équation dans [5] [3], que l'on a implémenté ici. L'intégration de cette unification dans l'algorithme d'unification usuel est directe : en effet, les unités sont des feuilles des types (autrement dit elles ne contiennent pas de sous-types) et n'introduisent donc pas de récursion. Lors de l'unification de deux types, il suffit donc d'unifier les couples d'unités un par un.

Algorithme d'unification dimensionnelle Soit une équation dimensionnelle $u_1 = u_2$. On commence par la mettre sous la forme $u_1 \cdot u_2^{-1} = 1$. Notons l'équation résultante

$$\delta_1^{x_1} \cdot \delta_2^{x_2} \cdots \delta_m^{x_m} \cdot b_1^{y_1} \cdot b_2^{y_2} \cdots b_n^{y_n} = 1$$

où les δ_i désignent les variables de dimension, et les b_j les dimension de base.

Si $m = n = 0$ il n'y a rien à faire. Si $m = 0$ et $n \neq 0$, il n'y a pas de solution et l'algorithme échoue.

Sinon, soit x_k l'exposant de plus petite valeur absolue parmi les x_i. Quitte à réordonner, on suppose que $k = 1$. Quitte à passer à l'inverse, on suppose aussi que $x_k \geq 0$.

À partir de ce point on distingue deux cas :
— Si $\forall i, x_i \bmod x_1 = 0$ et $\forall j, y_j \bmod x_1 = 0$, alors l'unificateur le plus général est le suivant :

$$\delta_1 \to \delta_2^{-x_2/x_1} \cdots \delta_m^{-x_m/x_1} \cdot b_1^{-y_1/x_1} \cdots b_n^{-y_n/x_1}$$

3. Il s'agit en fait d'une adaptation d'un algorithme plus général donné par Knuth dans [7], que l'on rencontrera par la suite.

— Sinon, on introduit la variable $\delta = \delta_1 \cdot \delta_2^{x_2/x_1} \cdots \delta_m^{x_m/x_1} \cdot b_1^{y_1/x_1} \cdots b_n^{y_n/x_1}$. On substitue δ_1 par son expression en fonction de d et des autres variables, de sorte que l'équation devient :

$$\delta^{x_1} \cdot \delta_2^{x_2 \bmod x_1} \cdots \delta_m^{x_m \bmod x_1} \cdot b_1^{y_1 \bmod x_1} \cdots b_n^{y_n \bmod x_1} = 1$$

À ce stade là, si δ est la seule variable restante, il n'existe pas d'unificateur et l'algorithme échoue. Sinon, on cherche à nouveau le plus petit exposant (en valeur absolue) et on réitère.

6. Comparaison, égalité

OCaml utilise un système de modules qui nécessite de pouvoir non seulement comparer des types mais aussi tester leur égalité, et ce sans copie ni modification. Les travaux existants n'intègrent pas de tel test, dont la nécessité est en partie liée aux choix d'implémentation d'OCaml (ces opérations pourraient être implémentées via l'unification, de façon moins efficace). Il a donc fallu développer une solution *ad hoc*.

On admet les définitions suivantes respectivement pour l'égalité et la comparaison de généralité entre deux types :

$$\tau_1 \sim \tau_2 \Leftrightarrow \exists \sigma, \sigma(\tau_1) = \tau_2 \text{ où } \sigma \text{ est un renommage des variables libres de } \tau_1 \text{ vers celles de } \tau_2$$

$$\tau_1 \leq \tau_2 \Leftrightarrow \exists \sigma, \sigma(\tau_1) = \tau_2 \text{ où } \sigma \text{ est une substitution des variables libres de } \tau_1$$

L'introduction d'unités modifie un peu la relation d'égalité : l'existence d'un renommage n'est plus une condition nécessaire pour que deux types soient égaux. Pour s'en convaincre, il suffit de considérer les types suivants :

— `<'a> dfloat -> <'a * 'b> dfloat`

— `<'c * m> dfloat -> <'d> dfloat`

Ces types sont bien égaux, et en particulier le premier n'est pas moins général que le second : en effet étant donnée une variable δ, toute unité peut être mise sous la forme d'un produit $\delta \cdot u$. Ils ne sont pourtant pas un renommage l'un de l'autre. Le critère d'égalité est ici le suivant : on a une substitution $[\alpha \to \gamma \cdot m\ ;\ \beta \to \delta \cdot \gamma^{-1} \cdot m^{-1}]$, qui va du premier au second type, et une substitution inverse $[\gamma \to \alpha \cdot m^{-1}\ ;\ \delta \to \alpha \cdot \beta]$ qui va du second au premier, toutes deux à coefficients entiers.

Les algorithmes implémentant égalité et comparaison en OCaml étant par nature structurels, plutôt que tenter d'y intégrer le traitement des unités, nous avons choisi de rassembler les équations correspondant aux unités pendant le parcours simultané des deux types, pour les résoudre à la fin, dans une approche similaire à celle de Wand [8]. Dans le cas de la comparaison $\tau_1 \leq \tau_2$, la résolution a pour inconnues les variables dimensionnelles de τ_1, ce qui nous donne une substitution des variables de τ_1 vers celles de τ_2. Pour l'égalité $\tau_1 \sim \tau_2$, on part du même système linéaire, qu'on doit cette fois-ci résoudre deux fois, pour les variables dimensionnelles de τ_1 et τ_2 respectivement, ce qui nous donne les deux substitutions inverses.

Autrement dit, on peut utiliser le même algorithme pour traiter les équations issues de l'égalité et de la comparaison. Pour des raisons de lisibilité on passe maintenant en notation matricielle pour représenter les équations : les coefficients de la matrice sont les exposants, leur colonne indique à quelle variable ou unité de base ils correspondent, et une ligne correspond à une équation. Comme dans l'unification on regroupe les deux membres.

Le système à résoudre pour une comparaison $\tau \leq \tau'$ se représente par une matrice en trois blocs :

$$\left(\begin{array}{c|c|c} G & D & B \end{array} \right)$$

où les blocs G, D, et B sont respectivement constitués des exposants des variables de τ, des variables de τ', et des unités de base.

Voici la matrice de l'exemple précédent, et sa normalisation pour (α, β) et (γ, δ).

$$\begin{pmatrix} \alpha & \beta & \gamma & \delta & m \\ 1 & 0 & -1 & 0 & -1 \\ 1 & 1 & 0 & -1 & 0 \end{pmatrix} \qquad \begin{pmatrix} \alpha & \beta & \gamma & \delta & m \\ 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 1 & -1 & 1 \end{pmatrix} \qquad \begin{pmatrix} \gamma & \delta & \alpha & \beta & m \\ 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & -1 & -1 & 0 \end{pmatrix}$$

Nous avons indiqué le titre de chaque colonne dans les matrices. Chaque ligne de la matrice problème (à gauche) correspond à l'égalité entre deux unités : la première entre `<'a> dfloat` et `<'c * m> dfloat`, et la seconde entre `<'a * 'b> dfloat` et `<'d> dfloat`.

Pour normaliser, on applique le même algorithme que pour l'unification, mais en reportant les substitutions effectuées sur chaque ligne de la matrice, plutôt que de modifier les types. Cela correspond précisément à l'algorithme originel de Knuth[7].

Si l'algorithme réussit, la matrice est sous une forme normalisée où :
— chaque colonne du bloc G a un coefficient à 1 et les autres à 0 ;
— si une ligne ne contient pas de 1 dans G, elle est nulle.
Sinon le test a échoué.

7. Perspectives

L'implémentation décrite ici est encore un prototype. Elle suffit à typer et exécuter les exemples en section 3, mais l'algèbre des types d'OCaml étant particulièrement riche, certains traits ne sont pas complètement compatibles avec l'usage des unités, en particulier la quantification universelle explicite.

Cependant, cette implémentation démontre que l'ajout d'une telle extension au typeur d'OCaml peut se faire de façon extrêmement légère : seules 53 lignes ont été modifiées ou ajoutées dans `ctype.ml`, le cœur du typeur, alors que la logique a pu être intégralement implémentée dans un module séparé, appelé seulement quatre fois depuis `ctype.ml`. Il serait intéressant de voir si en OCaml, comme en Haskell [4], il serait possible de transformer cette extension en un simple greffon.

De ce point de vue, le choix de n'étendre que la syntaxe des types, sans introduire de support particulier pour les termes, a payé. Il faudra plus d'expérience pour savoir si cela suffit en pratique.

Références

[1] B. Blanchet. Inférence de types avec dimensions sous Caml Light, 1996. `http://prosecco.gforge.inria.fr/personal/bblanche/cldim.html`.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[3] J. Goubault. Inférence d'unités physiques en ML. In *Journées Françaises des Langages Applicatifs*, pages 3–20, 1994.

[4] A. Gundry. A typechecker plugin for units of measure. In *Haskell Symposium*. ACM Press, 2015.

[5] A. Kennedy. Dimension types. In *Proc. 5th European Symposium on Programming : Programming Languages and Systems*, pages 348–362, 1994.

[6] A. Kennedy. Types for units-of-measure : Theory and practice. In *Proc. Third Summer School Conference on Central European Functional Programming School*, pages 268–305, 2010.

[7] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.) : Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[8] M. Wand and P. O'Keefe. Automatic dimensional inference. In *Computational Logic : in honor J. Alan Robinson*, pages 479–486. MIT Press, 1991.