Inférence d'unités physiques en ML

Jean Goubault[1,2]

1 *Bull coordination recherche*
rue Jean Jaurès
78 340 Les Clayes sous Bois, France
Jean.Goubault@frcl.bull.fr
2 *DMI-LIENS Ecole Normale Supérieure*
45, rue d'Ulm 75230 Paris CEDEX 05, France

Résumé : Nous décrivons une extension du système de types de Standard ML permettant un typage plus fin des quantités numériques, par l'incorporation de la notion de grandeur physique (masse, longueur, etc.). Le système est extensible, pleinement polymorphe, et effectue la vérification et l'inférence automatique des équations aux dimensions. Les unités physiques (kg, m, etc.) sont alors des échelles le long des dimensions, et le compilateur génère automatiquement les instructions de conversion entre différentes unités.

Nous en décrivons les principes, la réalisation et les extensions possibles ou souhaitables.

Mots clefs : ML, typage, inférence de types, dimensions physiques, unités de mesure.

1. Introduction

La plupart des langages de programmation fournissent un seul type de données numériques. Standard ML n'est pas une exception de ce point de vue, en ne fournissant que `real`. La plupart des applications de calcul numérique sont donc effectivement contraintes, même en ML, à être écrites dans un langage pratiquement non typé : tous les nombres ont le même type `real`.

Pourtant, les formules écrites dans ces applications, étant le plus souvent des formules physiques, obéissent à des conditions de cohérence que ne permettent pas de vérifier les systèmes de types usuels. Ces conditions sont appelées *équations aux dimensions* par les physiciens, et consistent à assurer que l'on n'additionne ni ne compare jamais deux valeurs de nature (de grandeur, de dimension : distances, temps, etc.) non comparables. Le contrôle de cohérence ainsi obtenu est en fait tellement fort qu'il est parfois possible d'inférer une loi physique uniquement à partir d'une équation aux dimensions[1]. La sécurité fournie par la vérification des équations aux dimensions est donc très grande.

Une telle vérification peut être automatisée ; il s'agit d'un problème de vérification de types. Ces types ne sont pas de même nature que les types de

[1] Un exemple éminent est la découverte des lois de la relativité générale (ou théorie de la gravitation) à partir de la relativité restreinte, fondée sur le fait que la gravitation et l'accélération sont une même grandeur.

données usuels des langages de programmation, et nous proposons d'enrichir le système de types de ML pour effectuer ces vérifications. Pour obtenir un système de types aussi souple que celui de ML, nous effectuons en fait l'*inférence* des grandeurs physiques, au moyen de types numériques polymorphes. Le système de types de ML étendu conserve ainsi les caractéristiques d'uniformité du typage, d'extensibilité (par le biais de déclarations de types) et d'expressivité (grâce au polymorphisme) du langage originel.

Qui parle de dimensions parle d'unités associées, et nous montrons comment combiner la vérification de grandeurs avec la compilation de conversions d'unités physiques, sans consommer de ressources à l'exécution.

Le plan de l'article est le suivant : en section 2, nous commençons par rappeler les principaux travaux antérieurs concernant les unités physiques dans les langages de programmation. En section 3, nous exposons les principes sous-tendant l'usage des grandeurs et des unités physiques, définissons les objets sémantiques, et illustrons l'usage typique de ces notions à travers d'exemples écrits en HimML [3], une extension de ML dans laquelle nous avons réalisé le système. La section 4 décrit la modification du système de types à réaliser : la nouvelle algèbre des types, et l'algorithme d'unification des types sont présentés. La section 5 décrit la façon dont les conversions d'unités sont compilées, et la section 6 décrit les points techniques principaux de la réalisation. En section 7, nous exposons les extensions ou améliorations souhaitables, avant de conclure en section 8.

2. Précédents travaux

Un des premiers travaux sur l'incorporation des unités physiques dans un langage de programmation (Pascal) est dû à Gehani [2], qui considère les unités physiques comme des attributs des types de données. Ces attributs peuvent être attachés au type `REAL` des réels en Pascal, et Gehani prétend que ces attributs peuvent entièrement être gérés au moment de la compilation.

Gehani confond, tout comme Karr et Loveman [7], le concept d'unités physiques (qui sont des échelles de mesure) et celui de grandeurs physiques (qui représentent la nature des mesures). L'intérêt principal de l'article de Karr et Loveman, qui se veut indépendant d'un langage particulier (encore que le style Fortran soit sensible), est la simplification des notions. Les auteurs montrent en particulier que la gestion des grandeurs physiques et des conversions entre unités se ramènent à de l'algèbre linéaire. Nous verrons en effet en section 3.1 que l'espace des grandeurs forme un espace vectoriel G sur un corps K, et en section 3.2 que celui des valeurs numériques est l'espace vectoriel $K \times G$.

House [6] fut le premier à noter la confusion fondamentale entre grandeurs et unités. Il remarqua de plus que les propositions de Gehani étaient inutilisables, ses algorithmes étant incorrects et nécessitant des vérifications d'unités physiques à l'exécution. House propose donc un système dans lequel ces derniers défauts sont corrigés, mais estime que la distinction entre grandeurs et unités n'a pas lieu d'être, à cause de la similarité des opérations disponibles sur les deux objets. Ceci est maladroit, car le système de types échoue alors

à fournir une documentation précise de la nature des objets du programme ; cette maladresse se ressent dans le besoin qu'a House d'ajouter à côté de chaque déclaration d'unité un commentaire indiquant la grandeur physique qui lui correspond.

Le système de House est encore une extension du système de types de Pascal, dont il estime qu'elle se transporte à Ada. Cette extension est sophistiquée, et introduit un niveau de polymorphisme explicite qui lui permet d'avoir un système correct même en présence de procédures passées en argument à d'autres procédures. Le polymorphisme est de plus, comme le remarque House, indispensable si l'on veut que le système d'unités physiques soit utilisable en pratique ; il serait insupportable de devoir réécrire une résolution d'équation linéaire pour toutes les grandeurs possibles des variables et des coefficients, par exemple. La gestion de ce polymorphisme est cependant malaisée en Pascal, et le système de types n'étant pas spécifié formellement, il est difficile d'être sûr qu'il reste correct dans tous les cas.

Männer [8] reconnaît, lui, que la distinction entre grandeurs et unités est importante. Mais l'usage de Pascal ou d'Ada rend difficile l'introduction d'unités comme liées à des grandeurs, et Männer préfère introduire les grandeurs comme des types intervalles construits à partir des unités. Ainsi, au lieu de concevoir les distances comme des grandeurs pouvant être mesurées en mètres, en années-lumières ou en fermis, il permet de définir par exemple l' unité `km`, puis de déclarer que le type distance est le type `0 km..MaxDistance`. Ceci semble une restriction inutile sur la nature des distances.

La discipline de type de ML, plus propre que celle de Pascal, est un cadre qui nous semble plus adapté pour développer un système de vérification et d'inférence de grandeurs — ce que les physiciens appellent résoudre les équations aux dimensions —, et d'en profiter pour fournir un système de conversion d'unités. Le système de type résultant, pleinement polymorphe et ne nécessitant que des adaptations minimes du typage de ML, est décrit dans les sections suivantes et a été réalisé en HimML [3].

On pourra trouver d'autres références dans les articles mentionnés. A l'heure où nous finissons d'écrire ces lignes, nous avons appris qu'Andrew Kennedy, de l'université de Cambridge, travaillait sur un projet similaire. Un rapport devrait sortir fin septembre 1993 sur le sujet. Nous n'avons pas encore connaissance de la teneur exacte de ce travail, mais il semblerait que ses idées soient proches des nôtres. De façon surprenante, il ne semble pas que le problème de la gestion automatique des grandeurs et unités physiques soit un sujet qui ait inspiré un grand nombre de travaux, ce qui explique le manque de références plus récentes sur le sujet.

3. Principes

Dans la suite, nous appelons *nombres* les éléments d'un corps commutatif K, qui sera le corps $\mathbb{R}$ des réels ou le corps $\mathbb{C}$ des complexes. Nous adoptons la convention selon laquelle $x^0 = 1$, même pour $x = 0$. De plus, nous considérerons

que les entiers sont des éléments de K, ce qui nous permet d'écrire[2] 3 au lieu de 3.0.

Appelons, informellement pour l'instant, *valeur numérique* tout nombre décoré d'un produit éventuellement vide de puissances d'unités de base, comme il est l'usage de représenter les valeurs de quantités physiques. Ainsi, 1.5, $10\mu F$ ou $9.81 m/s^2$ sont-elles des valeurs numériques.

La gestion des valeurs numériques avec unités repose sur les deux principes suivants :

- il est possible de convertir entre certaines unités ; par exemple, $1 mile = 1609.344 m$;
- l'addition, la comparaison de deux valeurs représentant des grandeurs de natures différentes n'a pas de sens ; par exemple, $1 mile + 1 kg$ n'a aucun sens.

Nous définissons d'abord les notions sémantiques de grandeurs, d'unités, puis nous procédons à l'illustration de leur utilisation sur un exemple en HimML. La manière formelle dont sont présentées ces notions a pour but d'établir la correction de l'algorithme d'unification des types (section 4.2), mais la compréhesion des développements formels n'est pas nécessaire à celle des principes et concepts.

3.1. Grandeurs

Pour gérer le deuxième point ci-dessus, un système de types à la ML est une façon d'écarter les expressions n'ayant pas de sens parce que représentant des sommes ou des comparaisons de grandeurs différentes. Les grandeurs de base en physique sont les distances, les masses, les temps, et les intensités électriques (nous ne nous limiterons cependant pas aux grandeurs *physiques* dans la suite). Toutes les autres grandeurs sont des produits de puissances de ces grandeurs de base : par exemple, la grandeur "force" est le produit de "masse", "distance" et "temps" à la puissance -2. Ainsi, les grandeurs forment-elles une algèbre[3], contenant la grandeur triviale *num* des valeurs sans dimension, et stable par les opérations de produit $\otimes$ des grandeurs, et de mise à la puissance $^\wedge$ d'une grandeur par un exposant.

L'ensemble des exposants, muni de l'addition et de la multiplication, est un anneau unitaire commutatif contenant tous les entiers relatifs (les exposants de "masse", "distance" et "temps" dans la grandeur "force" sont 1, 1 et -2 respectivement, par exemple). Cependant, il est aussi utile de pouvoir inverser des exposants. Par exemple, la racine carrée d'une distance a pour grandeur "distance"$^{1/2}$. C'est pourquoi nous choisissons le corps K comme ensemble des exposants.

[2] En HimML, il n'y a pas — à l'heure actuelle en tous cas — d'entiers, et la notation 3 produit en fait le réel 3.0.

[3] au sens logique du terme, c'est-à-dire un ensemble muni d'opérations obéissant à des lois équationnelles.

Ces considérations font de l'espace des grandeurs un espace vectoriel G sur K, dont l'addition est $\otimes$, et la multiplication externe est la mise à la puissance $\hat{}$. Une façon de s'en convaincre est, comme Karr et Loveman [7], de raisonner sur les logarithmes des grandeurs. L'impossibilité d'ajouter ou de comparer des grandeurs physiques de base est reflétée dans leur indépendance linéaire. Par exemple, ("masse","distance","temps") forme un système de vecteurs linéairement indépendants, dans lequel "force" a pour coordonnées $(1, 1, -2)$.

L'algèbre des types de ML est l'algèbre libre au-dessus des variables de types, engendrée par les symboles de fonctions (ou *noms*) de types déclarés par `datatype`, ou prédéfinis comme `string` ou `->`. L'algèbre des grandeurs physiques n'est pas libre, elle obéit aux équations définissant les espaces vectoriels. L'algèbre des types de ML ne peut donc pas simuler celle des grandeurs, et nous proposons de l'étendre dans ce but, en section 4.

3.2. Unités

Il est important, comme le remarque [6], de bien distinguer les concepts de *grandeur* physique et d'*unité* physique. Ce fait, bien connu des physiciens, a été pourtant ignoré par les premiers auteurs à proposer des systèmes d'unités physiques en informatique [2, 7]. Si les grandeurs représentent autant de natures différentes des valeurs, et permettent un contrôle de cohérence des expressions numériques du programme, les unités sont des échelles, des graduations, à l'intérieur de chaque grandeur.

Ainsi, au lieu de disposer d'un type numérique comme `real`, censé donner une représentation unique du corps $\mathbb{R}$ des réels, nous considérons autant de représentations du corps de base K qu'il y a de grandeurs physiques. Une *valeur numérique* est donc un couple formé d'un nombre et d'une grandeur physique, et l'espace des valeurs numériques est $N = K \times G$.

Une *unité*, ou *échelle*, pour la grandeur $g \in G$ est un point de repère sur la droite $K \times \{g\}$, différent de $(0, g)$, par rapport auquel toutes les valeurs de grandeur g sont mesurées : si l'unité u pour la grandeur g est le couple (x, g), $x \neq 0$, (par exemple, $u = km$ et $g =$"distance"), alors $3,5km$ représente le point $(3,5x, g)$. Notons que rien ne distingue les unités des valeurs numériques, les deux n'étant autres que des points de N. Ceci n'est pas un défaut ; en fait, les physiciens se servent de cette remarque. Ainsi, les vitesses de particules dans les accélérateurs sont-elles mesurées en c, où $1c$ est la vitesse de la lumière, et les chimistes mesurent-ils les quantités de produits en moles (avec $1mol = 6,023.10^{23}$, nombre d'Avogadro, sans dimension en fait).

Les opérations de base sur les valeurs numériques sont définies comme suit :

- $(x, g) + (y, g) = (x + y, g)$, et $(x, g) + (y, g')$ est indéfini si $g \neq g'$ (similairement pour la soustraction) ;
- $-(x, g) = (-x, g)$;
- $(x, g).(x', g') = (x.x', g \otimes g')$ (similairement pour la division) ;

- $(x,g)^e = (x^e, g^{\wedge} e)$.

Quant aux opérations transcendantes (log, exp, sin, etc.), elles s'expriment sous forme de séries entières $\sum_{i=0}^{+\infty} a_i (x,g)^e$ avec une infinité de a_i non nuls, ce qui n'est garanti correct d'après les lois ci-dessus que lorsque $g = num$. Nous conviendrons donc que ces fonctions sont de type $num \rightarrow num$. La mise à la puissance est un cas particulier : il est cohérent avec les règles ci-dessus de définir la fonction puissance par $(x,g)^{(y,num)} = (x^y, g^{\wedge} y)$ si $y \in K$, $(x,g)^{(y,g')}$ étant indéfini si $g' \neq num$. Ceci, malheureusement, oblige à injecter dans le typage (calcul de $g^{\wedge} y$) une quantité (y) qui ne sera connue qu'à l'exécution. Nous reviendrons sur ce problème en section 6.

Finalement, les opérations de comparaison comme $\leq$ ne sont définies que sur des valeurs de même grandeur ($(x,g) \leq (x',g)$ étant défini par $x \leq x'$), la valeur absolue est définie par $|(x,g)| = (|x|, g)$, et les autres fonctions numériques, comme la partie entière, ne s'appliquent qu'à des valeurs sans dimension, c'est-à-dire de grandeur num.

3.3. Illustration

Les grandeurs sont les dénotations des *types numériques*. Le plus simple est `num`, correspondant à la grandeur num. Les autres types numériques de base sont fabriqués par la déclaration `quantity`, créant une nouvelle grandeur, indépendante de toutes les précédentes. Mais toute grandeur $g \in G$ dispose d'une unité par défaut implicite, à savoir $(1,g)$. Nous proposons au programmeur de lui donner un nom lors de la déclaration de la quantité. Ainsi :

```
quantity distance(m)
```

déclare-t-elle `distance` comme un nouveau type numérique, d'unité par défaut `m` (le mètre). Pour recréer syntaxiquement l'algèbre des grandeurs, nous introduisons un opérateur ` de multiplication, et un opérateur ^ de mise à une puissance constante[4]. On peut ainsi écrire :

```
quantity temps(s)
type vitesse = distance`temps^ ~1
type acceleration = vitesse`temps^ ~1
```

Les unités sont manipulables par les mêmes opérateurs que les grandeurs. Ainsi, l'on peut déclarer de nouvelles unités :

```
scale 1`km = 1000`m (* 1 kilometre vaut 1000 metres [distance] *)
scale 1`h = 3600`s (* 1 heure vaut 3600 secondes [temps] *)
scale 1`kph = 1`km`h^ ~1 (* 1 kilometre par heure [vitesse] *)
scale 1`inch = 0.0254`m (* pouce anglais [distance] *)
scale 72.27`didot = 1`inch (* point didot [distance] *)
```

et demander à évaluer `3.5`h + 23`s`, ce qui aboutit au résultat `12623`s`. A contrario, `1`h+1`km` produira une erreur de typage (types numériques `temps`

[4] les notations, maladroites certes, sont le résultat d'une recherche de syntaxe n'entrant pas en conflit avec le reste de celle de Standard ML.

et `distance` non unifiables).

On peut se demander avec raison pourquoi nous avons choisi d'étendre la syntaxe, plutôt que de réutiliser les opérations arithmétiques de ML pour déclarer les produits et puissances d'unités. La raison est double. Premièrement, il fallait déjà étendre la syntaxe pour exprimer les produits et puissances de grandeurs. Ensuite, étendre la syntaxe permet de voir une valeur numérique affectée d'une unité comme une constante (par exemple `60‘kph`) utilisable dans les patterns ML, ce qui n'est pas possible avec des expressions de la forme `60 * kph`.

Nous proposons, plus généralement, et dans l'esprit de ML, un typage polymorphe des valeurs numériques. Ceci est réalisé par l'introduction de *variables de types numériques*, et l'algorithme de typage infère toujours le type principal si l'expression est typable. Les variables de types numériques sont notées avec un dièse après l'apostrophe commençant le nom de la variable, comme `'#a`, `'#b`, etc. Ce polymorphisme permet de conserver le statut de citoyen de première classe aux fonctions mathématiques usuelles. Ainsi, le type principal de `+` est-il `'#a * '#a -> '#a`, le type de `*`, `'#a * '#b -> '#a ‘ '#b`, et celui de `/`, `'#a * '#b -> '#a ‘ '#b^ ~1`.

Notons que si la discipline de type numérique est très stricte, la conversion d'une unité en une autre est néanmoins possible, en multipliant par une constante du type désiré. Par exemple, si x : `distance` est exprimé en `km`, et que l'on veuille exprimer x en secondes, il suffit de calculer x `* 1 s‘km^ ~1`.

Finalement, notons aussi que les unités et grandeurs ne sont pas limitées aux seules grandeurs et unités physiques. Une nouvelle grandeur peut être déclarée par `quantity` juste pour demander au langage de vérifier des équations aux dimensions. On pourra ainsi écrire `quantity apples(apples) and oranges(oranges)` pour déclarer deux nouvelles grandeurs incomparables, nommées `apples` et `oranges`, ou déclarer `quantity memoire(octet)` pour s'assurer que toutes les opérations de comptage de la mémoire ne comptent pas en fait autre chose. Il est alors par exemple possible de compter la mémoire en pages, et de déclarer la taille des pages localement par un `let scale 1‘page = 1024‘octet in...end`, cette déclaration étant, comme toutes les déclarations en Standard ML, à portée lexicale. Les unités permettent aussi de déclarer des unités sans dimensions, comme des multiplicateurs. Pour témoin, on peut écrire `scale 1‘deci = 0.1`, `scale 1‘dm = 1‘deci‘m`.

Le lecteur curieux pourra se demander pourquoi nous avons choisi ML, et en particulier Standard ML, comme point de départ de notre étude. En effet, le système est adaptable à n'importe quel langage possédant une discipline de type à la Hindley-Milner. Historiquement, l'auteur a, par goût personnel, choisi de réaliser un interprète de Standard ML dans le but de l'utiliser comme langage embarqué dans un tableur scientifique. Il était alors intéressant d'intégrer la gestion d'unités et de grandeurs physiques dans le langage même. Mais tout autre variante de ML, de Miranda ou de Haskell par exemple, auraient été des choix valides.

4. Typage

Les changements à effectuer en ML pour obtenir les constructions décrites dans la section précédentes sont minimaux. En ce qui concerne la sémantique statique, nous nous contentons de modifier l'algèbre des expressions de types, l'algorithme d'unification associé, et le type des opérations numériques. Tout le reste (unification des types, généralisation et instanciation des schémas de types) est inchangé.

4.1. Types

Soit $TyVar$ l'ensemble infini dénombrable des variables de type, et donnons-nous deux sortes `type` (la sorte des types) et $\texttt{type}^\#$ (la sorte des types numériques), avec $\texttt{type}^\# \subset \texttt{type}$. Notons $TyVar^\#$ le sous-ensemble de $TyVar$ des variables de sorte $\texttt{type}^\#$. Nous supposons que $TyVar^\#$ et $TyVar \setminus TyVar^\#$ sont tous les deux infinis, de sorte que nous disposons d'un réservoir illimité de variables des deux sortes. La construction est similaire à la définition des variables de types admettant l'égalité en Standard ML, où les types admettant l'égalité forment une sous-sorte de `type` [5, 10][5].

Soit $TyName$ l'ensemble des noms de types, et $\alpha : TyName \to \mathbb{N}$ une fonction d'arité. Notamment, les noms de types ML de base et leurs arités sont donnés par $\alpha(\texttt{string}) = 0$, $\alpha(\texttt{->}) = 2$, $\alpha(\texttt{ref}) = 1$, etc. La syntaxe des types ML est une syntaxe pour les termes du premier ordre sur $TyName$ et $TyVars$:

$Type$	::=	$TyVar$	
		$TyName$	si d'arité 0
		$Type\ TyName$	si d'arité 1
		$(Type, \ldots, Type)\ TyName$	si d'arité égale à la longueur du n-uplet de types

La syntaxe des types ML avec types numériques est la même que sans. Seul le vocabulaire de noms de types est changé : `real` disparaît, et `num` (d'arité 0), ‘ (infixe, d'arité 2), et tous les $\hat{}x$ (d'arité 1), pour $x \in K$, sont introduits.

L'algèbre des types est une algèbre à sortes ordonnées. Nous définissons une relation ::, où $\tau :: S$ signifie que le type τ est de sorte S. Chaque nom de type f possède une signature de la forme $S_1 \times S_2 \times \ldots \times S_n \to S$, avec $n = \alpha(f)$, ce que nous écrivons $f :: S_1 \times S_2 \times \ldots \times S_n \to S$. Considérant f comme du sucre syntaxique pour $()f$ (si $n = 0$), et τf comme du sucre syntaxique pour $(\tau)f$ (si $n = 1$), les règles d'attribution de sortes sont :

$$\frac{\tau :: S \quad S \subset S'}{\tau :: S'} \qquad \frac{\tau_1 :: S_1 \quad \ldots \quad \tau_n :: S_n \quad f :: S_1 \times S_2 \times \ldots \times S_n \to S}{(\tau_1, \ldots, \tau_n)f :: S}$$

et les signatures de base sont :

- $\texttt{num} ::\rightarrow \texttt{type}^\#$, ‘ $:: \texttt{type}^\# \times \texttt{type}^\# \to \texttt{type}^\#$, $\hat{}x :: \texttt{type}^\# \to \texttt{type}^\#$;

[5] les auteurs n'ont toutefois pas adopté cette présentation.

- pour tous les autres noms de types f prédéfinis, d'arité n, f :: $\underbrace{\texttt{type} \times \ldots \times \ldots \texttt{type}}_{n \text{ fois}} \rightarrow \texttt{type}$;
- pour tout nom de type f déclarée par `datatype`[6] d'arité n, f :: $\underbrace{\texttt{type} \times \ldots \times \ldots \texttt{type}}_{n \text{ fois}} \rightarrow \texttt{type}$;
- pour toute fonction de type déclarée par `quantity`, f ::$\rightarrow$ $\texttt{type}^{\#}$ (autrement dit, on ne peut déclarer que des fonctions de types numériques zéro-aires).

Si τ est un type, alors soit il n'a aucune sorte, et nous dirons qu'il est mal formé, soit il existe une sorte minimale parmi les sortes de τ. Il existe des types mal formés (par exemple, `string ^ 2`), et ceci est une différence majeure d'avec Standard ML, où tous les types syntaxiquement corrects sont bien formés. Nous reviendrons là-dessus en section 4.3, et supposons en attendant que tous les types que nous manipulons sont bien formés.

Nous appelons *type numérique* tout type de sorte minimale $\texttt{type}^{\#}$. Les types numériques sont `num`, les variables de types de $TyVar^{\#}$, les types f déclarés par `quantity`, et les produits et puissances de types numériques.

Syntaxiquement, nous notons `'a`, `'b`, ..., les variables de $TyVar \setminus TyVar^{\#}$, `'#a`, `'#b`, ..., celles de $TyVar^{\#}$. Les opérations arithmétiques de base sont alors (entre autres) `+ : '#a * '#a -> '#a`, `- : '#a * '#a -> '#a`, `~ : '#a -> '#a`, `* : '#a * '#b -> '#a ' '#b`, `/ : '#a * '#b -> '#a ' '#b ^ ~1`, `abs : '#a -> '#a`, `log : num -> num`, `floor : num -> num`, etc.

4.2. Unification dans l'algèbre étendue

Nous notons les substitutions σ ou $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$. Si τ est un type, σ et σ' deux substitutions, $\tau\sigma$ est τ substitué par σ, et $\sigma\sigma'$ est la composition de σ et σ', de sorte que $\tau(\sigma\sigma') = (\tau\sigma)\sigma'$. Nous omettons donc les parenthèses et écrivons $\tau\sigma\sigma'$.

L'espace des sortes muni de l'ordre strict $\subset$ est un cas particulier de forêt[7]. Un résultat standard de la théorie de l'unification est que dans ce cas, deux termes (ou types) bien formés ont un unificateur syntaxique le plus général, ou ne sont pas unifiables [11], comme dans le cas sans sortes.

Le problème ici est un peu plus compliqué, car nous opérons une unification sémantique, modulo la théorie des grandeurs physiques. Cette théorie n'affectant que les types numériques, nous pouvons construire une structure de données spécialisée pour les représenter, et changer ainsi la représentation interne des types en exploitant la structure d'espace vectoriel des grandeurs. Nous appelons donc dorénavant *type*, ou une variable de type non numérique α,

[6] Noter que `type` ne déclare pas des noms de types, seulement des abréviations de types.
[7] en fait d'arbre.

ou une application $(\tau_1, \ldots, \tau_n)f$, avec $f :: \underbrace{\texttt{type} \times \ldots \times \texttt{type}}_{n \text{ fois}} \to \texttt{type}$, ou un type numérique ; un *type numérique* est représenté par une mappe [4], ou fonction ensembliste de domaine fini, de $TyVar^\# \cup Q$ vers $K \setminus \{0\}$, où Q est l'ensemble des *quantités*, c'est-à-dire des noms de types de signature $\to \texttt{type}^\#$, autrement dit ceux déclarés par `quantity`. Nous faisons l'hypothèse qu'il est possible de distinguer les types numériques des autres types, par un système de tags par exemple.

Par exemple, supposons que `distance` et `temps` sont des quantités. Alors, le type des accélérations est noté `distance ' temps ^ ~2`, et correspond au type numérique $\{\texttt{distance} \mapsto 1, \texttt{temps} \mapsto -2\}$: un type numérique associe à chaque quantité qui le compose son exposant dans le type. En particulier, si `distance` est une quantité, il est aussi la notation pour un type numérique, à savoir $\{\texttt{distance} \mapsto 1\}$, où la quantité `distance` intervient avec l'exposant 1, et aucune autre quantité n'intervient. Pour représenter les types numériques polymorphes, nous admettons que les variables de types numériques jouent le rôle de quantités : `'#a ' '#b^0.5` est donc représenté par $\{'\#\texttt{a} \mapsto 1, '\#\texttt{b} \mapsto 0.5\}$. La notion de substitution, cependant, ne correspond plus à un simple remplacement textuel, mais à une opération algébrique de "produit" de types numériques (correspondant à la somme dans l'espace vectoriel des grandeurs). Formalisons ceci.

Cette dernière représentation est une forme normale pour les types numériques. Posons, pour tout type numérique $\tau^\#$, vu comme une mappe, ou ensemble de mappelets $x \mapsto y$:

- $\tau^\# \uparrow = \lambda x \cdot \begin{cases} \tau^\#(x) & \text{si défini} \\ 0 & \text{sinon} \end{cases}$;
- réciproquement, si f est une application de $TyVar^\# \cup Q$ vers K, le type numérique $f \downarrow$ est $\{x \mapsto f(x) \mid x \in TyVar^\# \cup Q \cdot f(x) \neq 0\}$;
- le domaine de $\tau^\#$ est noté $\mathsf{Dom}\,\tau^\#$. Il s'agit de l'ensemble des variables $\alpha \in TyVar^\#$ telles que $\tau^\# \uparrow (\alpha) \neq 0$;
- si A est un ensemble de variables de types, l'effacement de A dans f est $A \lhd f = \lambda x \cdot \begin{cases} f(x) & \text{si } x \notin A \\ 0 & \text{si } x \in A \end{cases}$.

Nous définissons les opérations $\otimes$, $^\wedge$ et de substitution sur les types numériques, sans surprise, par :

- $\tau_1^\# \otimes \tau_2^\# = (\lambda x \cdot \tau_1^\# \uparrow (x).\tau_2^\# \uparrow (x)) \downarrow$,
- $\tau^{\#\wedge} e = (\lambda x \cdot (\tau^\# \uparrow (x))^e) \downarrow$,
- $\tau^\#[\alpha_1 \mapsto \tau_1^\#, \ldots, \alpha_n \mapsto \tau_n^\#] = (\{\alpha_1, \ldots, \alpha_n\} \lhd (\tau^\# \uparrow)) \downarrow \otimes \tau_1^{\#\wedge}(\tau^\# \uparrow (\alpha_1)) \otimes \ldots \otimes \tau_n^{\#\wedge}(\tau^\# \uparrow (\alpha_n))$.

Les types numériques clos (sans variables) sont en bijection avec les grandeurs, et cette bijection est en fait un isomorphisme pour les opérations $\otimes$ et $^\wedge$. Nous

(Delete)	$E \cup \{\tau \stackrel{?}{=} \tau\}, \sigma$	$\leadsto$	E, σ
(Decomp)	$E \cup \{(\tau_1, \ldots, \tau_m) f \stackrel{?}{=} (\tau'_1, \ldots, \tau'_m) f\}, \sigma$	$\leadsto$	$E \cup \{\tau_1 \stackrel{?}{=} \tau'_1, \ldots, \tau_m \stackrel{?}{=} \tau'_m\}, \sigma$
(Bind)	$E \cup \{\alpha \stackrel{?}{=} \tau\}, \sigma$ si α non libre dans τ	$\leadsto$	$E[\alpha \mapsto \tau], \sigma[\alpha \mapsto \tau]$
(Num)	$E \cup \{\tau^\# \stackrel{?}{=} \tau'^\#\}$ avec $\sigma^\# = unif^\#(\tau^\# \otimes \tau'^{\#\wedge}(-1))$ si $\tau^\#$ et $\tau^\#$ sont des types numériques et $unif^\#$ n'échoue pas	$\leadsto$	$E\sigma^\#, \sigma\sigma^\#$

Figure 1: Règles pour l'unification

considérons désormais que les grandeurs sont précisément les types numériques clos.

Une valuation de variables de types numériques est alors une substitution close, et la sémantique des types numériques associe à tout $\tau^\#$ dans une valuation σ la grandeur $\tau^\#$ substituée par σ. En effet, l'opération de substitution est définie de sorte que si $\tau^\# = \alpha_1{}^\wedge e_1 \otimes \ldots \otimes \alpha_n{}^\wedge e_n \otimes \tau'^\#$, où $\tau'^\#$ ne dépend pas des α_i, alors $\tau^\#[\alpha_1 \mapsto \tau_1^\#, \ldots, \alpha_n \mapsto \tau_n^\#] = \tau_1{}^\wedge e_1 \otimes \ldots \otimes \tau_n{}^\wedge e_n \otimes \tau'^\#$. La notion de substitution, comme il est l'usage en logique, reflète la sémantique.

Unifier deux types numériques $\tau_1^\#$ et $\tau_2^\#$, c'est trouver une substitution σ telle que $\tau_1^\#\sigma = \tau_2^\#\sigma$, ou encore telle que $(\tau_1^\# \otimes \tau_2^{\#\wedge}(-1))\sigma$ soit la mappe vide $\{\}$, c'est-à-dire `num`. Le problème se ramène donc à unifier un type numérique avec `num`. Nous avons :

Proposition 1 *Soit $\tau^\#$ un type numérique.*

- *si $\tau^\# = \{\}$, alors $\tau^\#$ et* `num` *sont unifiables, et la substitution vide $[]$ est l'unificateur le plus général ;*
- *sinon, si $\mathsf{Dom}\,\tau^\# = \emptyset$, alors $\tau^\#$ et* `num` *ne sont pas unifiables ;*
- *sinon, soit α une variable quelconque de $\mathsf{Dom}\,\tau^\#$. Alors $[\alpha \mapsto (\{\alpha\} \lhd \tau^\#)^\wedge(-1/\tau^\#(\alpha))]$ est un unificateur le plus général de $\tau^\#$ et* `num`.

Preuve : Les deux premiers cas sont faciles, reste le dernier. Pour clarifier les choses, notons $\tau^\# = \alpha^\wedge e \otimes \tau'^\#$, où $\alpha \notin \mathsf{Dom}\,\tau'^\#$ et $e \neq 0$. Nous devons prouver que $\sigma_0 = [\alpha \mapsto \tau'^{\#\wedge}(-1/e)]$ est un unificateur le plus général. Il s'agit bien d'un unificateur, vérifions que tous les autres en sont des instances. Supposons donc que l'on ait σ telle que $\tau^\#\sigma =$ `num`. Alors $(\alpha^\wedge e \otimes \tau'^\#)\sigma =$ `num`. L'application de substitution commute avec la multiplication et la mise à la puissance des types numériques, donc $\alpha\sigma = (\tau'^{\#\wedge}(-1/e))\sigma$. Ceci implique que la composée $\sigma_0\sigma$ est égale à σ, prouvant que σ est une instance de σ_0. □

Nous notons $unif^\#(\tau^\#)$ la procédure découlant de la construction ci-dessus pour unifier un type numérique $\tau^\#$ avec `num`.

De cette proposition, l'algorithme d'unification des types (pas seulement numériques) découle immédiatement. La figure 1 le présente à l'aide d'un

système de règles de transformations de multi-ensembles d'équations E entre types, et de substitutions σ, dans le style de [1]. Une équation $\tau \stackrel{?}{=} \tau'$ est considérée équivalente à $\tau' \stackrel{?}{=} \tau$. Nous notons $E\sigma$ le multi-ensemble des $\tau\sigma \stackrel{?}{=} \tau'\sigma$ pour $\tau \stackrel{?}{=} \tau' \in E$.

Pour unifier τ avec τ', nous initialisons le multi-ensemble E à $\{\tau \stackrel{?}{=} \tau'\}$ et la substitution σ à $[]$, et déclenchons les règles de la figure avec une stratégie équitable quelconque ; lorsque la procédure s'arrête, soit le multi-ensemble est non vide et l'unification a échoué, soit la substitution construite par les applications des règles **(Bind)** et **(Num)** est un unificateur le plus général. Notons au passage que si les types sont bien formés, les règles ne produisent que des termes et des substitutions bien formés. Il y a donc en particulier au plus un unificateur le plus général, c'est-à-dire que l'extension de ML avec des types numériques conserve la propriété de l'existence d'un type principal.

L'algorithme est très proche des algorithmes d'unification syntaxique. En fait, seule la règle **(Num)** a été ajoutée au premier algorithme de [1]. Les techniques standard de codage de ces règles s'appliquent : si l'on représente les unificateurs σ en forme triangulaire, et que l'on n'applique pas réellement les substitutions sur E et σ dans la règle **(Bind)**, il existe une stratégie pour laquelle l'unification est linéaire en la taille des termes [9] ; fondamentalement, elle consiste à appliquer **(Delete)** en priorité, et **(Bind)** et **(Num)** le plus tard possible.

Nous avons ignoré, par souci de simplicité, les autres attributs des variables de types en Standard ML [5, 10]. Il s'agit de l'attribut d'égalité (si une variable de type l'a, elle ne peut être instanciée que par un type admettant l'égalité) et de l'attribut d'impérativité (une variable impérative ne peut être instanciée que par des types impératifs). Ces attributs peuvent être non booléens (un entier mesurant la "force" dans le schéma des types faibles de MacQueen, par exemple). Dans tous les cas, ces attributs modifient la structure de l'univers des sortes. Cependant, ceci a peu d'effet sur l'algorithme ci-dessus, l'univers complet des sortes étant un produit cartésien des univers correspondant à chaque attribut.

4.3. Types mal formés

Nous avons raisonné dans la section précédente sur des types bien formés, ce qui nous a permis de les représenter sur une forme canonique. Encore faut-il vérifier qu'un type est bien formé, ou, de façon équivalente, vérifier la légalité des opérations de construction des types.

Ceci apparaît à trois endroits dans un compilateur ML étendu avec des types numériques : lorsqu'il s'agit de construire un type produit $\tau * \tau'$, de construire un type puissance $\tau \hat{} e$, et d'appliquer une substitution σ à un type τ.

Dans les deux premiers cas, il suffit de vérifier que les arguments sont des types numériques ; il s'agit typiquement d'une vérification de tags. Le dernier cas est réglé en imposant qu'une substitution bien formée respecte les sortes, c'est-à-dire ici qu'elle ne remplace une variable de type numérique que par

un type numérique, au moyen de la définition de la substitution sur les types numériques.

Par exemple, la déclaration :

```
type '#a carre = '#a^2
```

est légale, parce que `'#a` est un type numérique. Par contre,

```
type 'a carre = 'a^2
```

ne l'est pas. Il est donc en particulier impossible de produite le type absurde `string^2` par unification d'une instance de `'a` avec `string`. Dans le premier cas, ceci est interdit parce qu'il est impossible d'unifier `'#a` avec `string`, ce dernier n'étant pas un type numérique.

Ceci demande de modifier la sémantique des déclarations de types ML. En effet, en Standard ML, les variables de types liées (comme `'a` ou `'#a` ci-dessus) voient leurs sortes ignorées. Les deux déclarations ci-dessus seraient alors équivalentes. Le fait de ne pas les considérer équivalentes, et de faire des abréviations de types et des noms de types des fonctions partielles de types ne demande que peu de changement à Standard ML, et suffit pour effectuer les vérifications de types illustrées ci-dessus. En particulier, les algorithmes de typage ne sont aucunement affectés.

5. Compilation

En ce qui concerne la sémantique dynamique, toutes les opérations de conversion d'unités sont des multiplications par des constantes, et peuvent être compilées par modification de l'arbre syntaxique en sortie de la phase d'inférence des types.

Ainsi, si l'on déclare :

```
quantity distance(m)
scale 1`km = 1000`m
```

l'expression `3.5`km` sera-t-elle modifiée (par un patch sur la syntaxe abstraite opéré par l'inféreur de types), en `3.5 * 1000`. Pour ce faire, le système d'inférence de types maintient une mappe dans l'environnement de typage, associant toute unité (c'est-à-dire toute échelle, déclarée par `scale`) à un couple $(x, \tau^{\#})$, $\tau^{\#}$ étant le type monomorphe associé à l'unité, et $x$ étant le quotient de l'unité à l'unité par défaut du type $\tau^{\#}$ (dans le cas d'un type composé comme `distance`temps^ ~2`, cette unité par défaut est la composée des unités par défaut ; par exemple ici, `m`s^ ~2`). Dans l'exemple ci-dessus, cette mappe serait $\{\texttt{m} \mapsto (1, \texttt{distance}), \texttt{km} \mapsto (1000, \texttt{distance})\}$. Il s'agit d'un recodage direct de la notion sémantique d'unité, telle que présentée en section 3.2.

Lorsque l'algorithme de typage rencontre l'arbre syntaxique correspondant à une constante numérique xu, où x est un nombre et u une unité de type numérique (monomorphe, nécessairement) g, il recherche le couple $(x', \tau^{\#})$ associé, patche l'arbre syntaxique en un nœud décrivant la multiplication de x par x', et déclare que l'expression possède le type $\tau^{\#}$.

Un autre problème est l'impression des valeurs numériques, en particulier au toplevel. Nous exploitons pour cela le fait que l'environnement ML connaît les types des données à imprimer, et nous imprimons les résultats numériques suivis de l'unité par défaut de leur type principal. Le nombre devant l'unité ne pose pas de problème à imprimer, puisqu'il s'agit de la valeur retournée par l'évaluateur, à cause de la façon dont nous laissons le système de types modifier l'arbre de syntaxe abstraite.

L'unité par défaut, elle, se retrouve à partir des noms des unités par défaut des quantités composant le type principal de la valeur numérique. Notons que ce type principal est toujours monomorphe, ce qui assure l'existence d'une unité par défaut. Pour coder cela, au lieu d'associer dans l'environnement de types, à tout nom de type f, un couple $(tyfun, cons)$ comme en Standard ML [5], où $tyfun$ est une fonction de type et $cons$ une description des constructeurs associés au nom de type, nous associons un triplet $(tyfun, cons, unit)$, où le champ additionnel $unit$ est le nom d'une unité par défaut ou un marqueur $-$. Les cas autorisés sont : $unit \neq -$, $cons = \emptyset$, $tyfun = f$ (quantité numérique), $unit = -$, $cons \neq \emptyset$, $tyfun = f$ (nom de type, déclaré par `datatype`), ou $unit = -$, $cons = \emptyset$ (abréviation de type déclarée par `type`). Les règles de typage de [5] doivent alors être modifiées ; ceci n'entraîne qu'un remplacement des couples $(tyfun, cons)$ par des triplets dans les règles concernées, et aucun changement profond.

6. Réalisation

En pratique, le corps K n'est pas codable de façon fidèle, et nous recourons à une approximation de ce corps. House [6] estime qu'il est indispensable, au contraire, de disposer de représentant exacts des éléments du corps des exposants. Ceci est effectivement faisable en général, tous les exposants étant en fait rationnels. Néanmoins, nous avons choisi en HimML une solution que l'on pourrait qualifier de facilité, consistant à utiliser des nombres flottants pour approcher les réels[8].

Certaines précautions, pour éviter de perdre de la précision dans les calculs, sont alors nécessaires. L'unification des types est essentiellement une technique d'élimination de variable dans les résolutions d'équations matricielles, comme le remarquent Karr et Loveman [7]. Le fait que nous ayons le choix dans l'algorithme $unif^{\#}$ entre toutes les variables du domaine du type numérique $\tau^{\#}$ correspond au fait que nous pouvons choisir n'importe quelle variable à éliminer pour résoudre un système d'équations linéaires, pourvu qu'elle apparaisse avec un coefficient non nul. Pour éviter de perdre trop de précision, nous opérons donc comme dans la méthode de Gauss, et choisissons la variable ayant le plus gros exposant en module, c'est-à-dire α telle que $\alpha \in \mathsf{Dom}\,\tau^{\#}$ et $|\tau^{\#}(\alpha)|$ est maximal.

En pratique, le fait que nous ne raisonnions que sur des approximations de nombres de K implique que le test d'égalité des types sous-tendant la procédure

[8] pour être exact, les nombres en HimML sont des complexes, au vu de leur utilité dans nombre de calculs ; ils sont représentés par des couples de flottants, ou de simples flottants si la partie imaginaire est nulle, les deux cas étant distinguables à l'exécution à l'aide de tags.

d'unification peut échouer à cause d'une erreur d'arrondi. Cependant, en général les exposants sont faibles, et l'expérience démontre qu'à condition d'utiliser un arrondi correct (comme dans la norme IEEE 754/954), l'erreur d'arrondi est en réalité nulle. House avait déjà remarqué que les exposants étaient petits en pratique, mais n'avait pas réalisé que cela faisait des flottants une représentation acceptable des exposants.

Finalement, notons que si le système de typage présenté ici est suffisamment propre, il faut quand même le salir un peu dans une implémentation, pour arriver à typer raisonnablement l'opération de mise à la puissance `**`. En général, `**` a pour type le plus général `num * num -> num`. Mais si nous faisons cela, la fonction de racine carrée `fn x => x**0.5` aura pour type le plus général `num -> num`, alors que le type `'#a -> '#a^0.5` était correct. Une solution est de définir, pour chaque constante $e \in K$, une opération spécifique de mise à la puissance e, de type `'#a -> '#a^`e. Il est en fait tout aussi simple de la noter `**`e, et de laisser l'inféreur de types reconnaître cette forme spéciale. Il doit cependant prendre des précautions, et interdire cette exception au système de types lorsque `**` a été redéfinie.

7. Extensions futures et questions

Une première extension du système proposé ici est l'adaptation au système de modules de Standard ML. En ce qui concerne le typage, le système de modules repose sur une version affaiblie d'une unification du second ordre pour régler les équations de partage (`sharing`)[9], les noms de types étant désormais variables lorsqu'ils sont déclarés dans des signatures. L'affaiblissement a pour but d'en obtenir une version décidable et produisant au plus une unificateur le plus général. Ainsi, deux schémas de types ne sont jamais unifiés que lorsqu'ils sont tous les deux des noms de types, variables ou non.

Dans le cas des types numériques, cette restriction est trop forte. Nous pouvons désirer, en effet, que le produit de deux types numériques d'une structure soit identique au produit de deux types numériques d'une autre structure, menant à des équations de partage de la forme `sharing` $\tau = \tau'$, où τ et τ' sont des schémas de types numériques, et non seulement des noms de types numériques. Ceci, heureusement, ne pose aucun problème particulier, puisque les quantités sont zéro-aires : les noms de quantités variables (du second ordre) se comportent donc comme des variables (du premier ordre), et l'unification du second ordre des schémas de types est soluble par l'algorithme $unif^{\#}$, si l'on considère maintenant les noms de quantités variables comme des variables.

Il est aussi nécessaire de pouvoir écrire des équations de partage entre unités, une fois que les types numériques correspondant vérifient une équation de partage. Ainsi, il doit être possible d'écrire :

```
signature A = sig
```

[9] et les passages de structures en argument aux foncteurs, qui sont essentiellement des équations de partage.

```
    quantity qA(uA)
    quantity qA'(uA')
    scale 1`u = 3`uA`uA'
  end

signature B = sig
    quantity qB(uB)
    scale 1`v = 25`uB
  end

signature C = sig
    structure A1 : A
    structure B1 : B
    sharing type qA`qA' = qB
    sharing scale 1`A1.u = 1`B1.v
  end
```

ce qui impose une contrainte sur les unités par défaut `uA` de `qA`, `uA'` de `qA'` en provenance de `A1`, et `uB` de `qB` en provenance de `B1`. Il semble que ceci soit faisable, au vu de la ressemblance entre unités et grandeurs, en ajoutant cependant une vérification de cohérence : nous devons nous assurer en effet que toute quantité physique a exactement une unité par défaut. Ceci aboutit à refuser une déclaration comme :

```
signature A = sig
    quantity q(u)
  end

signature B = sig
    quantity q(u)
  end

signature C = sig
    structure A1 : A
    structure B1 : B
    sharing type A1.q = B1.q
    sharing scale 1`A1.u = 2`B1.u
  end
```

Notons que cette proposition n'est pas entièrement satisfaisante, puisque la déclaration `quantity` dans les signatures ne permet de déclarer que des *quantités* monomorphes, et non des types numériques quelconques, en particulier polymorphes. Ainsi, on ne pourra pas déclarer `type 'a f` et `sharing type f^2=g^3` dans les signatures. Ces considérations devront être résolues dans une extension du système de modules aux unités physiques.

Un problème plus prosaïque est qu'il existe certains systèmes d'unités que nous ne pouvons pas traiter. Le cas typique est celui des températures, que l'on peut mesurer en Kelvin (K), ou en degrés Celsius, Fahrenheit ou Réaumur, sans qu'il y ait de loi de proportionnalité entre ces échelles. Karr et Loveman

ont déjà soulevé ce problème, et House le résoud dans le cas du langage Pascal en introduisant des unités convertibles par transformations affines, et non plus linéaires. Il est sans doute possible d'étendre le système présenté ici pour traiter ce genre de transformation. Le gain d'expressivité à en tirer semble cependant faible. De plus, ceci ne règle pas le problème des unités ayant une relation non affine avec une autre, comme le décibel et le Watt (un niveau de bruit est un logarithme d'une puissance).

Un problème plus grave est celui de l'impression des valeurs numériques. A l'heure actuelle, l'impression d'une valeur s'effectue toujours dans l'unité par défaut du type numérique de la valeur. Il est donc impossible d'afficher une distance d en `km` si l'unité par défaut des distances est le mètre `m`, à moins d'afficher le nombre sans dimension d `* 1'm'km^ ~1`, suivi de la chaîne de caractères `"'km"`. Ceci est cependant maladroit, et il serait intéressant d'y trouver une parade.

Finalement, nous revenons sur un problème que nous avons brièvement mentionné dans la note 2, celui de l'inexistence d'entiers à proprement parler en HimML, tous les nombres étant représentés en flottant. Si l'on souhaite disposer d'entiers, que ce soit pour l'efficacité des opérations (entiers machine) ou pour l'absence de perte de précision (entiers avec vérification de débordement, ou entiers multiprécision), nous pouvons choisir soit de considérer les entiers et les flottants comme deux représentations d'un même type, soit d'en faire deux types incompatibles `int` et `num` (`real` en Standard ML), avec des fonctions de conversion explicites. La première solution est en général préférée, en C, Pascal ou Scheme, mais elle mène à des problèmes de changement de représentations incontrôlés, qui peuvent induire des pertes de précision inattendues. Cet argument relativement spécieux est renforcé en présence d'une extension des réels à un système d'unités physiques : comment savoir si l'on effectue des calculs en entier ou en flottant lorsque le système peut choisir n'importe quelle échelle de base, faisant par exemple de la quantité apparemment entière `35 g` une expression stockée sous format flottant comme `0.35` (si l'unité par défaut est le `kg`) ? C'est pourquoi nous envisageons de couper court aux difficultés, et d'utiliser la seconde solution : bien qu'elle soit un peu plus malcommode pour l'utilisateur, elle a néanmoins l'avantage de le forcer à obéir à une discipline plus rigoureuse de gestion des types de nombres que lui offre la machine.

8. Conclusion

Nous avons décrit une extension du système de typage de Standard ML offrant un typage fin des valeurs numériques au moyen de la notion de types numériques. Un système de conversion d'unités est incorporé, qui est entièrement géré à la compilation, et n'introduit aucun ralentissement à l'exécution. Le système est extensible (nous pouvons à tout moment étendre le langage des quantités et des unités disponibles) et polymorphe, par l'usage de variables de types numériques. Le système est réalisé depuis fin 1991 à l'intérieur de HimML, et fonctionne rapidement et correctement.

Le système présente encore certains défauts, concernant notamment

l'impression des valeurs numériques et l'impossibilité de traiter des systèmes d'unités non linéaires. De plus, il est nécessaire de l'adapter au système de modules de Standard ML.

References

[1] Dershowitz (Nachum) et Jouannaud (Jean-Pierre). – Rewrite systems. *In : Handbook of Theoretical Computer Science*, éd. par van Leeuwen (Jan), chap. 6, pp. 243–320. – Elsevier Science Publishers b.v., 1990.

[2] Gehani (Narain). – Units of measure as a data attribute. *Computer Languages*, vol. 2, 1977, pp. 93–111.

[3] Goubault (Jean). – *The HimML Reference Manual.* – Rapport technique, rue Jean Jaurès, 78340 Les Clayes-sous-Bois, France, Bull Corporate Research Center, 1993.

[4] Goubault (Jean). – Une implémentation efficace de structures de données ensemblistes, fondée sur le hash-consing. *In : Journées francophones des langages applicatifs.* INRIA, pp. 222–238. – Annecy, février 1993.

[5] Harper (Robert), Milner (Robin) et Tofte (Mads). – *The Definition of Standard ML.* – MIT Press, 1990.

[6] House (R.T.). – A proposal for an extended form of type checking of expressions. *The Computer Journal*, vol. 26, n° 4, 1983, pp. 366–374.

[7] Karr (Michael) et Loveman (David B. III). – Incorporation of units into programming languages. *Communications of the ACM*, vol. 21, n° 5, May 1978, pp. 385–391.

[8] Männer (R.). – Strong typing and physical units. *SIGPLAN Notices*, vol. 21, n° 3, 1986, pp. 11–20.

[9] Martelli (A.) et Montanari (Ugo). – An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, vol. 4, n° 2, 1982, pp. 258–282.

[10] Milner (Robin) et Tofte (Mads). – *Commentary on Standard ML.* – MIT Press, 1991.

[11] Walther (Christoph). – Unification in many-sorted theories. *In : Proceedings of the 6th European Conference on Artificial Intelligence.* – Pisa, Italy, September 1984.