Projet d'informatique
Inférence de types avec dimensions sous Caml Light

Blanchet Bruno
Sous la direction de M. Roberto DiCosmo

9 juillet 1995

1 Introduction

Ce projet consiste à modifier le compilateur Caml Light, pour lui permettre de contrôler les dimensions des données manipulées. Les vérifications de type effectuées par le compilateur sont donc renforcées, et les erreurs peuvent être signalées à l'utilisateur.

Première partie

Utilisation du nouveau compilateur

Ce compilateur est basé sur Caml Light version 0.7 béta 1.

1 Syntaxe des dimensions

La grammaire des expressions dimensionnelles est définie ainsi (voir le fichier **parser.mly** pour tous les détails, ainsi que le fichier **dimen.ml** pour la conversion de l'arbre rendu par la grammaire en un arbre plus proche du programme) :

```
Dim : Dim * Dim
    | Dim / Dim
    | Dim ** Rat (* Elévation à un exposant rationnel *)
    | Identificateur de dimension
    | Variable de dimension
    | 1         (* Pour les données sans dimension *)
    | ( Dim )

(* Grammaire pour les nombres rationnels *)
Rat : Entier
    | Rat * Rat
    | Rat / Rat
    | - Rat
    | Rat - Rat
    | Rat + Rat
    | ( Rat )


Opt_dim_par :
       [ Dim (, Dim)* ]
     |

Dim_params :
      [ Variable de dimension (, Variable de dimension)* ]
     |
```

On peut donc effectuer sur les dimensions les opérations multiplicatives et l'élévation à un exposant rationnel. On a choisi des exposants rationnels plutôt qu'entiers, car ces derniers sont parfois insuffisants, par exemple pour les constantes de vitesse des réactions chimiques, ou pour les fonctions d'onde en mécanique quantique. De plus, les exposants rationnels sont plus faciles à implémenter que les exposants entiers.

Les variables de dimensions sont notées comme les variables de type ('identificateur).

Dans les déclarations de types, les paramètres dimensionnels viennent après les éventuelles paramètres de type et avant le nom du constructeur. (Ceci évite toute ambiguité pour savoir à quel type se rapporte le paramètre).

Les seuls constructeurs prédéfinis qui prennent des arguments dimensionnels sont `int` et `float`. L'utilisateur peut définir ses propres types dimensionnés à partir de ceux-ci. Par exemple :

```
type ['d] complex = {re : ['d] float; im: ['d] float};;

type 'a ['d] abbrev == 'a * ['d] complex;;
```

Les paramètres dimensionnels sont facultatifs. Si certains paramètres manquent, le comportement du compilateur varie suivant les options de compilation : voir ci-dessous.

2 Déclaration des dimensions

Tous les identificateurs de dimension doivent être déclarés par l'instruction

```
dimension nom(unite) ( and nom(unite) )* ;;
```

Exemple : `dimension length(m) ; ;`

Après cette déclaration, le mot `length` est utilisable dans les types. Par exemple : `type length_t == [length] float ; ;`

Le mot `m` est alors une constante de type `[length] float` et qui a pour valeur numérique 1. D'autres unités peuvent être déclarées en écrivant par exemple : `let km = 1000. *. m` . Si on souhaite utiliser des unités entières, il suffit de les convertir avec `int_of_float`.

L'unité entre parenthèses est facultative. Si on ne définit pas d'unité en créant la dimension, on peut en définir une plus tard en écrivant :

```
#open "dfloat";;
dimension length;;

let m = (fone : [length] float);;
```

(Voir plus loin la définition de `fone` dans le module **dfloat**).

Les déclarations de dimensions effectuées dans un module et destinées à être exportées doivent être faites dans la partie interface (fichier **.mli**). Elles seront automatiquement prises en compte dans l'implémentation, et ne doivent donc pas y être répétées.

Des déclarations classiques d'unités et de constantes physiques ont été enregistrées dans le module **dimens**.

3 La directive de compilation dim

Cette directive a été ajoutée pour contrôler le fonctionnement par défaut du compilateur. Elle comporte 4 options :

#dim "warning" Quand un paramètre dimensionnel manque, un message d'avertissement est affiché, mais la compilation continue.

#dim "nowarning" Le contraire du précédent : aucun message n'est émis, et la compilation se poursuit.

#dim "defnodim" Par défaut, les grandeurs sont sans dimension. Ainsi, quand un paramètre dimensionnel est absent, il est équivalent à `[1]`. Les constantes numériques sont sans dimension (`[1] int` ou `[1] float`). La variable de contrôle d'une boucle `for` est sans dimension. Les données dans les chaînes de formats sont sans dimension.

#dim "defanydim" Le contraire du précédent. Les grandeurs dont la dimension n'est pas spécifiée sont de dimension quelconque (`['d] int` ou `['d] float`).

L'état par défaut est `nowarning`, `defnodim`. Ceci est nécessaire pour obtenir une compilation sans faille des programmes réalisés pour l'ancienne version du compilateur. Par exemple, une construction comme

```
let x=ref 0
```

doit pouvoir être utilisée, ce qui implique que la constante entière 0 est sans dimension (les références polymorphes sont refusées). Pour que l'interface soit compatible avec l'implémentation, il faut alors que `int` et `float` soient par défaut sans dimension.

4 Modifications des librairies

Les anciennes librairies **float** et **int** ne sont plus adaptées pour les dimensions : elles ne sont capables de manipuler que des grandeurs sans dimension.

C'est pourquoi deux nouvelles librairies **dfloat** et **dint** ont été créées. Elles traitent correctement les dimensions.

Ces librairies définissent aussi de nouvelles fonctions spécifiques aux dimensions :

zero : Constante 0 de dimension quelconque. Utile pour définir des fonctions polymorphes (la valeur absolue par exemple).

one : Constante 1 de dimension quelconque. Elle permet de passer outre tout contrôle de dimension. A utiliser avec précaution donc.

nodim : Fonction à un argument qui assure que cet argument est sans dimension et le rend.

anydim : Fonction qui prend un argument de dimension quelconque, et rend la même valeur numérique, mais dans une autre dimension. Elle passe outre tout le contrôle de dimensions. A utiliser avec précaution donc.

Ces fonctions sont définies sous le nom indiqué dans le module **dint** et à la fois sous ce nom, et sous un deuxième nom obtenu en ajoutant un f au début dans le module **dfloat**. Ainsi en ouvrant dans l'ordre

```
#open "dfloat";;
#open "dint";;
```

on accède à la fonction entière par `zero`, `one`, ... et à la fonction flottante par `fzero`, `fone`, ... On peut avoir des noms plus simples pour les fonctions flottantes si on n'utilise qu'elles en réouvrant **dfloat** après **dint**.

J'ai choisi de dimensionner les fonctions logiques bit à bit ainsi :

- `lor` est dimensionnée additivement :

```
prefix lor : ['d] int -> ['d] int -> ['d] int
```

- `land` et `lxor` sont dimensionnées multiplicativement :

```
prefix land : ['d1] int -> ['d2] int -> ['d1 * 'd2] int
prefix lxor : ['d1] int -> ['d2] int -> ['d1 * 'd2] int
```

Ce choix est bien sûr discutable, mais je pense que de toute façon, il ne devrait pas être trop gênant, car ces fonctions sont utilisées essentiellement avec des grandeurs sans dimensions.

La fonction `power`, notée encore `prefix **` du module **float** pose des problèmes de dimensionnement : la dimension du résultat est fonction de la VALEUR de l'exposant, ce qui n'est normalement par toléré. De plus, pour que cela puisse être

correct, cet exposant doit être une constante rationnelle. On aurait pu faire un traitement spécial dans le compilateur, mais tout s'écroulerait en cas de redéfinition par l'utilisateur de **. Le choix effectué est le plus simple : `power` utilise normalement des arguments sans dimension. Si on veut traiter des arguments avec dimension, on a deux méthodes :

- Passer outre le contrôle des dimensions et poser

  ```
  #open "dfloat";;
  let (powera_b : ['d] float -> ['d ** (a/b)] float) =
    fun x -> fanydim(fanydim(x) ** (a. /. b.);;
  ```

  où a et b sont des constantes entières.

- Définir une fonction spécifique, dans le cas d'élévation à un exposant entier (dans ce cas, de toute façon, la méthode utilisant ** est très peu efficace). Par exemple :

  ```
  let sqr x = x * x;;
  let power5 x = sqr(sqr(x)) * x;;
  ```

  A cet effet, j'ai rajouté la définition de `sqr` dans le module **dfloat**, car je pense que cette fonction sera d'usage courant.

5 Contrôle des unités des entrées-sorties

Les fonctions `print_int`, `print_float`, `read_int`, `read_float` rendent ou prennent des valeurs sans dimension. Le programme doit fixer l'unité dans laquelle il veut la valeur.

Par exemple :

```
#open "dfloat";;
#open "dimens";;

print_string "Entrez la longueur en m :";
let x=read_float() *. m
in  print_string "La longueur en km est :";
    print_float (x /. km);;
```

On aurait aussi pu imaginer de fixer uniquement la dimension dans le programme, l'utilisateur entrant lui-même l'unité voulue. Il aurait alors fallu faire un analyseur syntaxique qui connaisse les différentes unités entrées dans le programme, et qui vérifie que la donnée entrée a bien la dimension voulue. Il faudrait alors une déclaration spécifique pour les unités dérivées, pour que le programme puisse les distinguer des constantes numériques ordinaires.

On pourrait faire un système analogue pour l'affichage, le programme choisisant lui-même l'unité à utiliser pour l'affichage. Un tel programme serait certes commode, mais assez compliqué à réaliser et contraire à l'esprit de Caml qui veut que le typage soit entièrement vérifié à la compilation.

Dans le cas particulier du toplevel, j'ai fait les modifications nécessaires pour que le programme affiche les données dimensionnées avec leur dimension dans le type, et l'unité associée avec la valeur, quand une unité a été définie avec la dimension.

Attention! Cette fonction d'affichage des unités ne respecte pas la syntaxe imposée à l'utilisateur pour la saisie de ces unités. En particulier, elle utilise ** avec les unités, alors que cette fonction est réservée aux grandeurs sans dimension. D'autre part, elle ne tient pas compte des modules ouverts : Elle écrit toujours * et

jamais *. que les grandeurs soient réelles ou entières, et que les modules **dfloat** et **dint** soient ouverts ou non. Elle utilise les unités flottantes, même si les grandeurs considérées sont entières. Quand elle ne peut déterminer l'unité voulue, soit parce que la grandeur est de dimension polymorphe (pour `zero` et `one`), ou que l'unité n'a pas été déclarée dans `dimension dim(unite) ; ;`, elle affiche `standard unit(s)` à la place. Par exemple :

```
##open "dimens";;
##open "dfloat";;
#m;;
- : [length] float = 1.0 * m
#sqrt(weber);;
- : [length * mass ** (1 / 2) * intensity ** (1 / -2) * time ** (-1)] float =
1.0 * m * kg ** (1 / 2) / A ** (1 / 2) / s
#int_of_float m;;
- : [length] int = 1 * m
#zero;;
- : ['a] float = 0.0 standard unit(s)
##open "dint";;
#dimension longueur;;
Dimension longueur defined.
#let long=100 * (one : [longueur] int);;
long : [longueur] int = 100 standard unit(s)
```

Cet affichage a été implanté juste pour que le système ait l'air correct du point de vue du physicien (les valeurs numériques doivent toujours être accompagnées de leurs unités), mais il reste beaucoup de progrès à faire pour obtenir quelque chose de tout à fait satisfaisant.

6 Exemples

On trouvera des exemples de programmes utilisant les dimensions dans le répertoire **~blanchet/projet/test**. Ces exemples ont essentiellement pour but de tester les fonctionnalités du compilateur. Ils n'ont donc rien de réaliste. En voici un petit :

```
#open "dimens";;
#open "dfloat";;
#open "printf";;

let calcenerg v m =
   printf "L'energie de masse de la particule est %.3g MeV\n" (m * sqr(c) / MeV );
   let mu = m / sqrt(1.-sqr(v/c)) in
   printf "Son energie cinetique est %.3g MeV\n" ((mu - m) * sqr(c) / MeV);
   printf "Son energie totale est %.3g MeV\n" (mu * sqr(c) / MeV);
;;

let exemples =
   print_string "\nElectron a la vitesse c/2 :\n";
   calcenerg (c/2.) me;
   print_string "\nProton a la vitesse c/2 :\n";
   calcenerg (c/2.) mp;
   print_string "\nNeutron a la vitesse c/2 :\n";
   calcenerg (c/2.) mn
;;

let saisie =
   print_string "\nEntrez la masse de la particule (en kg) : ";
   let m=read_float() * kg in
      print_string "Entrez sa vitesse (en fraction de c) : ";
      let v=read_float() * c in
         calcenerg v m
;;

exemples;;
flush std_out;;
saisie;;
flush std_out;;
```

Deuxième partie

Algorithmes utilisés

Les algorithmes utilisés pour le contrôle des dimensions sont particulièrement simples.

1 Format interne des dimensions

Le type des dimensions est déclaré dans **globals.ml** de la façon suivante :

```
(* Dimensions *)

type dimtyp_desc = DConst of string global | DVar of mutable dlink

and dimtyp = {dim_desc : dimtyp_desc;
              mutable dim_level:  int}

and dlink = Dnolink | Dlinkto of dimtype

and dimtype == (dimtyp * int * int) list;;    (* exposants rationnels *)
```

Les variables sont représentées par des liens, ce qui permet de faire une unification destructive. Une dimension `dimtype` est représentée comme une liste de dimensions élémentaires `dimtyp` accompagnées d'un exposant rationnel non nul. Les variables de dimension apparaissent dans cette liste systématiquement avant les constantes dimensionnelles, ce qui simplifie certaines fonctions.

Les identificateurs de dimensions sont représentés par un enregistrement

```
 string global == {qualid : qualified_identifier;
                                    (* Nom de la dimension *)
                   info : string    (* Unité associée, "" si aucune *)
                  }
```

Cette structure est aussi celle utilisée pour le stockage dans l'interface des modules. De plus, elle permet de retrouver facilement l'unité associée à une dimension pour les affichages dans le toplevel.

2 Unification

L'unification est réalisée dans **dimen1.ml**

```
let dim_unify dim dim'=
   let dim2=dim_type_repr(mult (-1) 1 dim dim') in
      match dim2 with
        [] -> ()
      | (({dim_desc = DVar (Dnolink as v)},n,m) ::dim3) ->
         v <- Dlinkto (expo (-m) n dim3)
      | _ ->
         raise Unify
;;
```

La fonction `mult n m dim dim'` calcule $dim^{(n/m)} * dim'$ et `expo n m dim` calcule $dim^{(n/m)}$ (Voir **dimen0.ml**). `dim_type_repr` calcule la forme instanciée d'une dimension, en remplaçant les variables instanciées par leur valeur.

On commence donc par calculer le quotient des deux dimensions à unifier. On a alors trois cas possibles :

- Le quotient est sans dimension : les deux dimensions de départ sont identiques.
- Le quotient contient une variable : on peut le rendre sans dimension en substituant convenablement cette variable, et ainsi unifier les deux dimensions de départ.
- Sinon, on ne peut pas unifier les dimensions.

3 Filtrage

Le filtrage est aussi réalisé dans **dimen1.ml**

```
(* Trouve une variable v qui est dans dim' et pas dim
   Lève l'exception Unify s'il n'y a pas de telle variable *)

let rec dim_occur_check dim =
   fun (({ dim_desc = DVar(Dnolink) as v} ,n,m) as t1 :: l) ->
          if occur v dim then
              let (t2,l2)=dim_occur_check dim l
              in (t2,t1::l)
          else
              (t1,l)
    | _ -> raise Unify
      (* Quand on rencontre une constante, il n'y a plus de variable *)
;;

(* Instancie dim pour qu'elle soit égale à dim' *)

let dim_filter dim dim'=
   let dim2=dim_type_repr dim
   and dim2'=dim_type_repr dim' in
   if mult (-1) 1 dim2 dim2'=[] then
      ()
   else
      let ((ty,n,m),dim3)=dim_occur_check dim2' dim2 in
      match ty.dim_desc with
        DVar(v) -> v <- Dlinkto (expo m n (mult (-1) 1 dim3 dim2'))
      | _ -> raise Unify
;;
```

On commence par calculer le quotient des deux dimensions pour voir si elles sont identiques. Sinon, on cherche une variable apparaissant dans dim2 et pas dans dim2' pour pouvoir, en instanciant cette variable, rendre dim2 égal à dim2'. S'il n'y a pas de telle variable, c'est l'échec.

Le filtrage est utilisé pour vérifier la compatibilité de l'interface et de l'implémentation d'un module.

Troisième partie

Installation et accès aux sources

Le seul fichier à récupérer est **cl7dim.tar.gz**. En décomprimant cette archive, on crée le répertoire **cl7dim** qui correspond au répertoire **src** de la distribution **Caml Light**. Ensuite, il faut indiquer dans le **Makefile**, où trouver une distribution Caml-Light 0.7 compilée. (Pour éviter de transporter trop de fichiers, le système établit des liens vers cette distribution pour les fichiers inchangés). Il ne reste plus alors qu'à lancer

```
make world
```

Remarque : Ne pas utiliser inconsidérement `make clean`, car comme il n'y a pas d'installation dans un répertoire extérieur, cela efface des fichiers nécessaires au fonctionnement du système.

Le code source du compilateur est dans **compiler**, les librairies dans **lib** (seuls les fichiers d* sont de ma fabrication), le toplevel est dans **toplevel**. Le code exécutable est dans **camlcomp**, et les scripts de lancement s'appellent **camlc** et **camllight**. Leur utilisation est la même que pour Caml-Light 0.7. Si le répertoire **cl7dim**, ou celui de la distribution Caml-Light 0.7 n'est pas dans le path, il faut exécuter les programmes en appelant explicitement **camlrun** (Celui de la version 0.7, pas celui de la version 0.6 !).

Ces scripts et les fichiers **Makefile** de **cl7dim** et **cl7dim/launch** ont été modifiés pour qu'il ne soit pas nécessaire d'installer Caml dans les répertoires de binaires **/usr/local/bin**... , ce qui nécessite d'être root. Il faudra revenir au fonctionnement initial pour obtenir une distribution plus classique.

Références

Jean Goubault.

Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference In *Computational Logic : in honor of J. Alan Robinson* (J.-L.Lassez and G. D. Plotkin, eds), MIT Press, (1991), pages 479-486.

Kennedy. *Dimension types.* ESOP 1994 (to appear)

Table des matières