TP5 - Modules et types abstraits de données

(*UPS - L3 Informatique - UE PFITA*)

Consignes générales

Les exercices de ce TP5 doivent être traités dans l'ordre de l'énoncé. Ils portent sur la définition et l'utilisation de types abstraits de donnés en OCaml grâce aux *modules*, vus au dernier TD PFITA.

Pour rappel, OCaml impose que le nom des modules **débute par une majuscule** (par exemple : `INT`). De plus nous choisissons ici comme *convention* d'écrire les noms des *modules types* en majuscules, précédé par la lettre « t » (par exemple : `tARITH`).

Note : les énoncés notés avec une étoile comme **3)*** ne sont pas testés par la plateforme (il s'agit essentiellement de questions de réflexion ne nécessitant pas d'écriture de code, mais dont la réponse nécessite la validation de votre encadrant de TP !)

Exercice 1 - Définition d'une signature

Définir la signature `tARITH`, introduisant :

- un type `t` (abstrait)
- des valeurs `zero` et `one` (de type `t`)
- des opérations d'addition (`add`) et de multiplication (`mul`), agissant sur deux éléments de ce même type `t`
- une opération d'opposé (`opp`) prenant un argument de type `t`
- une fonction `of_int` de type `int -> t` convertissant un entier en un élément de type `t`
- une fonction `to_string` de type `t -> string` renvoyant une représentation textuelle des éléments de type `t`

Exercice 2 - Modules implémentant une signature

Le module type `tARITH` défini précédemment peut être utilisé pour implémenter différentes arithmétiques, que l'on pourra manipuler via la même « interface » (comprenant le type des éléments arithmétiques manipulés).

1) Écrire un module `INT` de signature `tARITH` définissant l'arithmétique sur les entiers natifs d'OCaml (type `int`).

Indication : Vous avez le droit d'utiliser la fonction standard `string_of_int`.

2) En définissant un type utilisateur approprié, écrire un module `M3` de signature `tARITH` définissant l'arithmétique modulo 3.

Indications :

- Vous pouvez réutiliser votre code du TP précédent (*types utilisateurs*) pour cette question.
- Les opérations considérées doivent vérifier les axiomes usuels de l'arithmétique.
- En particulier, l'opération d'opposé est une fonction à un argument vérifiant pour tout `(x:t)`, `add x (opp x) = zero` (**axiome A1**).

- Similairement, on aura pout tout `x:t`, `mul x one = x` (**axiome A2**) ;
  et pour tous `(m:int)` `(n:int)`, `add (of_int m) (of_int n) = of_int (m + n)` (**axiome A3**).
- Pour `of_int` : votre fonction pourra commencer par calculer `n mod 3`, en faisant attention au fait que `n mod 3 <= 0` si `n < 0`.
- La représentation textuelle (`to_string`) des constructeurs `Zero`, `Un`, `Deux` sera respectivement `"0"`, `"1"`, `"2"`.

3)* Les entiers natifs OCaml, comme ceux de beaucoup d'autres langages, correspondent aussi à une arithmétique modulaire.

De quel modulo s'agit-il, et pourquoi est-ce une arithmétique modulaire ? Vous pouvez vous aider de cette page de documentation) pour trouver des éléments de réponse.

4)* Après avoir implémenté les modules précédents `INT` et `M3` (terminés par `end`), expérimentez l'utilisation de ces modules :

- quel est le type de `M3.one` ? (vous pouvez utiliser le toplevel pour vérifier)
- écrire trois fonctions `(checkA1 : M3.t -> bool)`, `(checkA2 : M3.t -> bool)` et `(checkA3 : int -> int -> bool)` qui renvoient `true` si les axiomes A1, A2 et A3 ci-dessus sont vérifiés pour les valeurs données en paramètre.
- dans le toplevel, faites `open M3;;` pour pouvoir écrire `add` au lieu de `M3.add`,
- et définissez des constantes `two`, `three`, `six` à partir de `one`, `zero` et `add` et/ou `mul`.
- Testez les fonctions `checkA1` et `checkA2` sur les constantes `two`, `three`, `six`, puis `checkA3` sur plusieurs exemples d'entiers.

Exercice 3 - Définition d'une signature paramétrée

Les signatures peuvent être utilisées pour "paramétrer" des signatures ou des modules. Le mot *foncteur* est synonyme de *module paramétré*. Cet exercice 3 s'intéresse à définir la *signature d'un foncteur*, puis l'exercice 4 sera consacré à son *implémentation*.

Définir la signature `tEXP` d'un foncteur qui, partant d'un module `A` de signature `tARITH`, construit un type abstrait représentant des expressions, de manière analogue au type `exp` demandé dans le TP4 précédent. Ici, ce type des expressions sera nommé `t`, vous pourrez donc utiliser les types `A.t` (pour les constantes) et `t` (pour les expressions) pour déclarer la signature des différentes opérations.

Comme ce type `t` est abstrait (le type concret des expressions n'est pas défini dans la signature), la signature devra déclarer différentes opérations pour construire une expression :

- à partir d'une constante de type `A.t` (opération `cst`)
- en tant qu'opposé d'une expression (`opp`)
- en tant qu'opération sur deux expressions (`add` et `mul`)

Enfin, une opération `compute` décrira l'évaluation d'une expression pour obtenir sa valeur (de type `A.t`).

Exercice 4 - Foncteur implémentant une signature paramétrée

Écrire un foncteur `EXP` implémentant la signature `tEXP`. Ce foncteur sera donc paramétré lui-aussi par une implémentation de la signature `tARITH`.

Pour cela, on commencera par définir le type concret `t` d'une expression définie dans le module paramétré `EXP`. Cette définition de type doit utiliser des constructeurs, similairement au TP4, c'est-à-dire qu'une expression de type `t` est **soit** une constante de type `A.t`, **soit** l'opposé d'une expression de type `t`, **soit** une addition **ou** une multiplication de deux expressions de type `t`.

On implémentera alors les opérations `cst`, `add`, `opp` et `mul` en utilisant les précédents constructeurs du type `t`.

Enfin, on pourra écrire l'opération `compute` en utilisant le filtrage et les opérations appropriées provenant du module de type `tARITH`.

Exercice 5 - Instanciation d'un foncteur

Il est possible d'instancier le foncteur `EXP` avec chacun des deux modules de l'exercice 2, afin de construire et de calculer des expressions dans les deux arithmétiques que nous avons définies.

1)

On crée maintenant deux modules `EXP_INT` et `EXP_M3` pour traiter respectivement des expressions d'entiers ou de nombres modulo 3. Ces deux modules sont des instances du module paramétré `EXP` avec les modules `INT` et `M3` comme paramètres, ils sont créés grâce aux deux instructions suivantes :

```
module EXP_INT = EXP(INT)
module EXP_M3  = EXP(M3)
```

2) À l'aide de l'instance `EXP_INT`, définissez l'expression `expr_int` correspondant à `2 * (2 + 2)`.

Indication : en supposant que `Module` contient `champ0, champ1, champ2`, l'expression

```
Module.(champ2 (champ1 1) champ0)
```

est un raccourci pour l'expression équivalente suivante :

```
Module.champ2 (Module.champ1 1) Module.champ0
```

3) À l'aide de l'instance `EXP_M3`, définissez l'expression `expr_m3` correspondant à `2 * (2 + 2)`.

4)* En utilisant le *Toplevel*, affichez les valeurs `expr_int` et `expr_m3`. Que signifie le mot-clé `<abstr>` ? Comment pouvez-vous obtenir la valeur concrète correspondant à ces expressions ?

Exercice 6 - Les polynômes à 1 variable définis par un foncteur

Ce dernier exercice est un exercice "bonus", reprenant toutes les étapes des exercices précédents sur un exemple plus concret.

La signature `tARITH` (exercice 1) correspond, mathématiquement parlant, aux opérations d'une *structure d'anneau*.

Dans cet exercice, on se propose de définir un foncteur qui associe, à un anneau `A`, les opérations des polynômes à une variable à coefficients dans `A`.

1) Définissez une signature paramétrée `tPOLY` de ce foncteur. Celui-ci construit, en partant d'un module `A` (de signature `tARITH`), un type abstrait qui **étend** `tARITH` (c'est-à-dire qu'il doit posséder toutes les opérations de `tARITH` (`zero`, `one`, `add`, `mul`, `of_int`, `to_string`), pour des polynômes de type `t` ayant des coefficients de type `A.t`), avec les opérations supplémentaires suivantes :

- construction d'un polynôme constant (`cst`, dont la constante de type `A.t` est passée en paramètre)
- construction du polynôme simple qui correspond à `1 * X^1` (`varx`)
  (plus précisément, dans la représentation des polynômes par `a0 + a1 * X + a2 * X^2 + ...`, `varx` correspondra au polynôme dont le coefficient `a1 = 1` et où tous les autres `ai` valent `0`)
- évaluation d'un polynôme en un point (`evalx`)
- construction rapide d'un polynôme en fournissant un entier *n* (nombre de coefficients) et une fonction associant un coefficient à chaque degré `d < n` (opération `def`, de type `int -> (int -> A.t) -> t`)

2) Définissez un foncteur `POLY` implémentant `tPOLY`. L'opération `def n f` permettra donc de construire le polynôme correspondant à `f 0 + (f 1) * X + (f 2) * X^2 + ... + (f (n - 1)) * X^(n-1)`.

Comme souvent, il existe plusieurs manières de procéder à l'implémentation. Il est cependant suggéré de représenter un polynôme tel que `a0 + a1 * X + a2 * X^2` par une liste de ses coefficients `[a0; a1; a2]` (en n'imposant pas que `a2 <> 0`).

La fonction `of_int` prenant un entier en argument, renverra un polynôme constant contenant cet entier comme coefficient.

La fonction `to_string` produira une chaîne de caractères correspondant au polynôme écrit sous la forme dite de Horner. En particulier, le polynôme `11 * X + 2 * X^2` sera affiché sous la forme suivante :

```
"0 + x*(11 + x*(2))"
```

3) Créez le module `POLY_INT`, instanciant le foncteur `POLY` avec le module `INT`, puis utilisez le module obtenu pour définir les 2 expressions suivantes :

- `poly_eval2` correspondant à `1 + 2 + 2^2 + 2^3 ... + 2^7`
- `poly_eval3` correspondant à `1 + 3 + 3^2 + 3^3 ... + 3^7`

Pour ce faire, vous définirez d'abord le polynôme `poly7 = 1 + x + x^2 + x^3 + ... + x^7` que vous afficherez sous forme de `string` en utilisant la fonction appropriée du module `POLY_INT`.

Comme pour les autres questions, vous pouvez utiliser le *Toplevel* pour vous aider à élaborer votre réponse ou pour vérifier le type de vos expressions. (Pour définir `poly_eval2` et `poly_eval3`, essayez d'écrire le code le plus concis possible.)

4)* À partir du code déjà écrit, décrivez une méthode permettant de construire l'anneau des polynômes à 2 variables.