Programmation impérative et fonctionnelle avec OCaml
TP 7

Objectifs :
— Types algébriques
— Interprétation
— Foncteurs
— Modularité et abstraction de type

Interprétation abstraite

L'interprétation abstraite est une technique d'approximation de la sémantique d'un langage de programmation qui permet d'obtenir des propriétés sur un programme sans l'exécuter complètement. Avant de réaliser une interprétation abstraite, il est nécessaire de définir :
— un *domaine* d'interprétation ;
— une *interprétation* des valeurs et des opérateurs (les fonctions, les structures de contrôle...) du langage dans ce domaine.

Prenons un exemple : avec la sémantique classique, le type `int` est interprété dans l'ensemble des entiers naturels $\mathbb{N}$ (ou plus précisément dans l'intervalle $[-2^{62}, 2^{62}[$ pour les entiers d'OCaml) ; dans une interprétation abstraite pour laquelle on ne s'intéressera qu'au signe des expressions entières, on pourra se contenter d'une *approximation* des entiers $\mathbb{N}$ par l'ensemble à 3 valeurs $\mathcal{S} = \{-, ?, +\}$, où un entier négatif est interprété par $-$, un entier positif ou nul par $+$ et un entier dont on ne connaît pas le signe par ?.

Les fonctions opérant sur le type `int` doivent également être interprétées par une fonction à valeurs dans le domaine abstrait. Par exemple, la fonction d'addition `+` pourra être interprétée par une fonction définie de la manière suivante :

$$\begin{array}{rcl} \mathcal{S} \times \mathcal{S} & \longrightarrow & \mathcal{S} \\ (+,+) & \mapsto & + \\ (-,-) & \mapsto & - \\ \text{sinon} & \mapsto & ? \end{array}$$

On s'intéresse ici à des expressions sur des nombres flottants (dont le domaine d'interprétation classique est donc l'ensemble des réels $\mathbb{R}$). On ne considèrera que les expressions suivantes :
— constante ;
— variable ;
— somme et produit de deux expressions ;
— conditionnelle sur trois expressions `c`, `t` et `e`, qui s'évalue en `t` si `c`≥ 0 et en `e` sinon.

On va écrire un foncteur pour pouvoir interpréter ces expressions dans différents domaines :

1. Dans un fichier `abstract.ml`, écrire la signature `DOMAIN` des domaines d'interprétation qui doit comporter :
   — le type `t` des éléments du domaine d'interprétation ;
   — une fonction `str` qui prend un élément du domaine (de type `t`) en paramètre et renvoie une chaîne de caractères qui le représente ;
   — une fonction `cst` qui prend un flottant en paramètre et renvoie un élément du domaine ;
   — les fonctions `sum` et `mul` qui prennent deux éléments et en renvoie un ;
   — la fonction `cond` qui prend trois éléments et en renvoie un.
2. Écrire le type `t` permettant de représenter les expressions.
3. Pour que l'évaluation d'une expression soit générique vis-à-vis d'un domaine quelconque, déplacer le type `t` dans un foncteur `Make` qui prendra un module `Dom` de signature `DOMAIN` en paramètre. On y ajoutera le renommage `type dom = Dom.t` pour distinguer le type `t` du foncteur `Make` et celui du module `Dom`.

4. Dans le foncteur `Make`, écrire la fonction `eval` d'interprétation des expressions dans le domaine `Dom` qui prend un environnement (une liste de couples `string * dom`) et une expression en paramètres et renvoie un élément de type `dom`. Pour chaque constructeur du type `t`, cette fonction appelera la fonction correspondante du module `Dom`, sauf dans le cas d'une variable où on utilisera `List.assoc` pour consulter l'environnement.
5. Dans un fichier `tp7.ml`, écrire le module `Sign` de signature `DOMAIN` du domaine des signes, puis appliquer le foncteur `Abstract.Make` à ce module.
6. Ouvrir (exceptionnellement !) le module ainsi obtenu et représenter l'expression suivante : `Si (x-y) Alors x Sinon (1+3)`
7. Tester l'évaluation sur l'expression précédente :
   — avec `x` positif et `y` quelconque, on doit obtenir $+$;
   — avec `x` négatif et `y` quelconque ou négatif, on doit obtenir ? ;
   — avec `x` négatif et `y` positif, on doit obtenir $+$.
8. On souhaite à présent abstraire la représentation des expressions (type `t` du foncteur `Make`), donc on ne pourra plus construire d'expressions directement à partir des constructeurs du type `t` (car ils seront cachés dans la signature du résultat du foncteur). Pour pouvoir construire des expressions, définir les fonctions suivantes à l'intérieur du foncteur :

```
val cst : float -> t
val var : string -> t
val ( + ) : t -> t -> t
val ( * ) : t -> t -> t
val cond : t -> t -> t -> t
```

9. Écrire la signature `S` du résultat de l'application du foncteur `Make` en y incluant tous les types et fonctions définis précédemment, mais en rendant abstrait le type `t` (et le type `dom` qui l'est ici forcément).
10. Dans un fichier `abstract.mli`, recopier les signatures `DOMAIN` et `S` puis déclarer le type du foncteur en restreignant son résultat à `S` avec la contrainte de type permettant d'utiliser les valeurs de type `dom`.
11. Tester votre implémentation en modifiant le fichier `tp7.ml` pour construire les expressions grâce aux fonctions et opérateurs exportés par le foncteur (à la place des constructeurs).
12. On considère maintenant l'ensemble $\mathcal{I}$ des intervalles de réels comme domaine d'interprétation. Un élément de $\mathcal{I}$ peut être représenté par le couple de flottants $[a, b]$ de ses bornes inférieure et supérieure, et les opérations arithmétiques pourront être approximées par arithmétique des intervalles. Écrire le module `Interval` de type `DOMAIN` du domaine $\mathcal{I}$, puis appliquer le foncteur `Abstract.Make` à ce module.
13. Ouvrir (l'exception devient la règle...) le module ainsi obtenu et représenter la même expression que précédemment puis tester l'évaluation sur l'expression précédente :
   — avec `x` et `y` dans $[10, 20]$, on doit obtenir $[4, 20]$;
   — avec `x` dans $[10, 20]$ et `y` dans $[2, 8]$, on doit obtenir $[10, 20]$.