Programmation en OCAML du noyau de Tarski's world

Mathieu Jaume
Mathieu.Jaume@lip6.fr

1 Termes du premier ordre

Définition des termes

La notion de termes permet de généraliser la plupart des objets manipulés en informatique : listes, piles, arbres, expressions, programmes, types, preuves ... Les termes sont les objets de base de la logique du premier ordre, utilisée pour exprimer des propriétés énoncées à l'aide de ces termes. L'ensemble $T_\Sigma[V]$ des termes est défini inductivement à partir :

- d'un ensemble V de symboles de variable
- d'une signature Σ c'est à dire d'un ensemble de symboles de fonction muni d'une fonction $ar : \Sigma \to \mathbb{N}$ d'arité[1]

$T_\Sigma[V]$ est défini inductivement comme suit :

- Un symbole de variable est un terme.
- Si $f \in \Sigma$ est un symbole d'arité n, et si $t_1, \cdots, t_n$ sont des termes, alors $f(t_1, \cdots, t_n)$ est un terme.

$$(\textsc{tv})\frac{}{x} \qquad (\textsc{tf})\frac{t_1 \cdots t_n}{f(t_1, \cdots, t_n)}$$

Le type OCaml pour représenter les termes peut être défini par :

```
type ('a,'b) term =   Var_term of 'a
                    | Cons_term of 'b * ((('a,'b) term) list);;
```

Il s'agit d'un type polymorphe où `'a` correspond au type des variables et `'b` au type des éléments de la signature Σ.

Remarque. Aucun contrôle n'est effectué sur le respect de l'arité des symboles de fonction : il n'est pas possible de définir des types dépendants en OCAML (comme par exemple le type des listes de longueur n) ... si l'on dispose de la fonction d'arité associée à Σ, il est toutefois possible de définir une fonction permettant de tester la “bonne formation” des termes.

Exemple. Les formules de la logique des propositions (en notation préfixe) peuvent être vues comme des termes construits à partir de la signature $\Sigma = \{\textsc{True}, \textsc{False}, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$[2] et d'un ensemble V de variables propositionnelles. En OCaml, on peut représenter le type des éléments de Σ par :

```
type connecteurs_logiques = CTrue|CFalse|CNeg|CAnd|COr|CImpl|CEquiv;;
```

[1] Pour $f \in \Sigma$, $ar(f)$ désigne le nombre d'arguments de la fonction f. Une constante peut être vue comme une fonction sans argument (d'arité 0).

[2] avec la fonction d'arité :

$$ar(f) = \begin{cases} 0 & \text{si } f \in \{\textsc{True}, \textsc{False}\} \\ 1 & \text{si } f \in \{\neg\} \\ 2 & \text{si } f \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \end{cases}$$

Par exemple, si l'on choisit d'indexer les variables propositionnelles par les entiers, la formule $\neg(p_1 \Leftrightarrow (p_1 \Rightarrow p_2))$ pourra être définie par le terme suivant :

```
let f = Cons_term(CNeg,
                  [Cons_term(CEquiv,
                            [Var_term(1);
                             Cons_term(CImpl,[Var_term(1);Var_term(2)])])]);;
(* val f : (int, connecteurs_logiques) term *)
```

Termes et substitutions

L'ensemble Θ des substitutions est l'ensemble des fonctions de V dans $T_\Sigma[V]$:

$$\Theta = \{\theta : V \to T_\Sigma[V]\}$$

On note s_{id} la substitution identité (pour tout $x \in V$, $s_{id}(x) = x$).

Souvent, les substitutions manipulées correspondent à l' "identité presque partout" : elles ne modifient qu'un ensemble fini de variables, appelé le domaine de la substitution :

$$dom(\theta) = \{x \in V \mid \theta(x) \neq x\}$$

Dans ce cas, on peut représenter une substitution par une liste d'association :

$$\theta = [(x_1, t_1), \cdots, (x_n, t_n)]$$

où t_i est le terme associé à la variable x_i (i.e. $\theta(x_i) = t_i$ et $\theta(y) = y$ pour tout $y \notin dom(\theta) = \{x_1, \cdots, x_n\}$). Il ne s'agit que d'un moyen de donner une représentation extensionnelle d'une fonction : θ est avant tout une fonction et est donc déterministe (i.e., pour toute variable x, $\theta(x)$ existe et est unique). La liste $[(x_1, t_1), \cdots, (x_n, t_n)]$ ne représente une substitution de domaine fini que si les variables x_1, ..., x_n sont deux à deux distinctes. Avec cette représentation des substitutions, la fonction qui permet d'appliquer une substitution à une variable s'écrit :

```
let rec apply_subst_V s x =
    try (List.assoc x s) with _ -> (Var_term x);;
(* apply_subst_V : ('a * ('a, 'b) term) list -> 'a -> ('a, 'b) term *)
```

L'application d'une substitution à un terme est définie de manière classique par[3] :

$$\hat{\theta} : T_\Sigma[V] \to T_\Sigma[V]$$
$$\hat{\theta}(t) = \begin{cases} \theta(x) & \text{si } t = x \in V \\ f(\hat{\theta}(t_1), \cdots, \hat{\theta}(t_n)) & \text{si } t = f(t_1, \cdots, t_n) \end{cases}$$

[3]On rappelle qu'une application $\varphi : T_\Sigma[V] \to T_\Sigma[V]$ est un morphisme ssi :

$$\varphi(f(t_1, \cdots, t_n)) = f(\varphi(t_1), \cdots, \varphi(t_n))$$

On montre alors que toute application $h : V \to T_\Sigma[V]$ se prolonge en un unique morphisme $\hat{h} : T_\Sigma[V] \to T_\Sigma[V]$. L'existence s'établit en définissant :

$$\hat{h}(t) = \begin{cases} h(x) & \text{si } t = x \in V \\ f(\hat{h}(t_1), \cdots, \hat{h}(t_n)) & \text{si } t = f(t_1, \cdots, t_n) \end{cases}$$

Pour montrer l'unicité, soit $\hat{h}_1$ et $\hat{h}_2$ 2 morphismes prolongeant h, montrons par induction sur t que $\hat{h}_1(t) = \hat{h}_2(t)$. Si $t = x \in V$, alors $\hat{h}_1(t) = \hat{h}_2(t) = h(x)$. Si $t = f(t_1, \cdots, t_n)$, alors, par hypothèse d'induction, on a $\hat{h}_1(t_i) = \hat{h}_2(t_i)$ et donc $\hat{h}_1(t) = f(\hat{h}_1(t_1), \cdots, \hat{h}_1(t_n)) = f(\hat{h}_2(t_1), \cdots, \hat{h}_2(t_n)) = \hat{h}_2(t)$.

ce qui se définit en OCaml par :

```
let rec apply_subst_T s t = match t with
   (Cons_term (f,lt))
      -> (Cons_term (f,(List.map (apply_subst_T s) lt)))
 | (Var_term x) -> (apply_subst_V s x);;
(* apply_subst_T:('a*('a,'b) term) list->('a,'b) term->('a,'b) term *)
```

Interprétation des termes

Interpréter un terme, c'est lui associer une valeur appartenant à un certain ensemble $\mathcal{D}$ appelé domaine d'interprétation. Pour pouvoir interpréter un terme, il faut associer une signification à chacun des symboles qui peuvent apparaître dans ce terme. Par construction, ces symboles appartiennent à l'ensemble $\Sigma \cup V$. L'interprétation des symboles de Σ s'obtient à partir d'une Σ-algèbre de domaine $\mathcal{D}$:

$$\mathcal{A}_\Sigma = (\mathcal{D}, \mathcal{I}_\Sigma)$$

Chaque symbole $f \in \Sigma$ est associé à une fonction $[\![f]\!]_{\mathcal{I}_\Sigma} : \mathcal{D}^{ar(f)} \to \mathcal{D}$. Les valeurs associées aux symboles de variable sont obtenues à partir d'une valuation $\nu : V \to \mathcal{D}$. Etant données une Σ-algèbre $\mathcal{A}_\Sigma$ de domaine $\mathcal{D}$ et une valuation $\nu : V \to \mathcal{D}$, le schéma d'interprétation des termes est défini par :

$$[\![t]\!]_{\mathcal{A}_\Sigma,\nu} = \begin{cases} \nu(x) & \text{si } t = x \in V \\ [\![f]\!]_{\mathcal{I}_\Sigma}([\![t_1]\!]_{\mathcal{A}_\Sigma,\nu}, \cdots, [\![t_n]\!]_{\mathcal{A}_\Sigma,\nu}) & \text{si } t = f(t_1, \cdots, t_n) \end{cases}$$

L'implantation de ce schéma en OCaml s'écrit donc :

```
let rec interp_term v interp_f t =
  match t with
    Var_term x -> (v x)
  | Cons_term(f,l) -> ((interp_f f) (List.map (interp_term v interp_f) l));;
(* interp_term: ('a->'b) -> ('c->'b list->'b) -> ('a,'c) term -> 'b *)
```

Langage de termes de Tarski's world

Le logiciel Tarski's world permet de "vérifier" des formules logiques du premier ordre exprimant des propriétés sur des objets géométriques placés sur une grille. Ces objets peuvent être associés à un nom : `a`, `b`, `c`, `d`, `e`, `f`. Pour simplifier, dans ce qui suit, on impose que tous les objets soient associés à un nom ... on se donne donc un ensemble de noms plus "grand" mais fini. Les objets sont des termes : ces noms correspondent à des constantes.

```
type constantes = A | B | C | D | E | F | G | H | I | J | K | L | M |
                  N | O | P | Q | R | S | T | U | V | W | X | Y | Z;;
```

Les seuls symboles de fonction sont les constantes (d'arité nulle) : il n'y a pas de fonction qui étant donné un ou plusieurs objets désigne un autre objet.

Enfin, les propriétés sont exprimées par des formules logiques du premier ordre : l'ensemble des variables disponibles est { `u`, `v`, `w`, `x`, `y`, `z` }. Dans ce qui suit on se donne un ensemble de variables indexé par les entiers :

```
type variables = VAR of int;;
```

Ainsi, `(VAR 0)`, `(VAR 1)`, `(VAR 2)`, ... sont des variables. L'ensemble des termes du langage de Tarski's world est donc :

$$T_{\texttt{constantes}}[\texttt{variables}]$$

Les termes (donc les noms d'objet) sont interprétés par des objets placés sur une grille. Cette grille est un ensemble de cases (repérées par des coordonnées) : une case est soit vide, soit contient un objet.

```
type 'a case = Empty | Case of 'a ;;
```

Chaque objet est associé à une forme et une taille :

```
type forme = Pyramide | Cube | Dodecaedre;;
type taille = Petit | Moyen | Grand;;
type objet = { form : forme ; dim : taille };;
```

Un “monde de Tarski” est donc un tableau de cases pouvant contenir des objets.

Un terme est interprété par une paire d'entiers correspondant aux coordonnées d'une case de la grille contenant un objet. Si aucun objet n'a été associé par l'utilisateur à un nom d'objet alors ce nom est interprété par une valeur spéciale : `Undef`.

```
type valeur = Obj of int*int | Undef;;
```

L'interprétation des symboles de fonction (donc des constantes, c'est à dire des noms d'objets) se fait étant donné un monde de Tarski. Afin de pouvoir utiliser la fonction `interp_term` définie plus haut, il faut définir une fonction de type $\Sigma \to \mathcal{D}$ `list` $\to \mathcal{D}$ qui permet d'interpréter les symboles de fonctions. Cela se fait simplement en considérant un monde de Tarski. Par exemple, à partir du monde ci-dessous :

		(A) △		

il suffit de définir la fonction :

```
let interp = function nom_cste -> function l -> match l with
    [] -> (match nom_cste with   A -> (Obj (2,2))
                               | _ -> Undef)
  | _  -> failwith "Arity error" ;;
```

Pour disposer de la forme d'un objet dont on dispose du nom, il suffira alors de définir la fonction :

```
let forme_de monde_tarski nom_cste =
   match (interp nom_cste []) with
     Obj (i,j) -> (match monde_tarski.(i).(j) with
                       (Case o) -> o.form
                     | Empty -> failwith "Undefined constant")
   | Undef -> failwith "Undefined constant" ;;
```

Exercice 1

1. A partir d'une grille, par exemple la grille vide que l'on peut obtenir à l'aide de la fonction :

```
let make_empty_world n m = Array.make_matrix n m Empty;;
```

on souhaite pouvoir construire un modèle en disposant des objets dessus. Définir deux fonctions `add_obj` et `sup_obj` qui permettent respectivement d'ajouter et de supprimer un objet sur la grille.

2. On veut pouvoir construire une interprétation des symboles de fonction en ajoutant ou en supprimant "l'association" d'une constante à un objet sur la grille à partir d'une interprétation donnée.
   (a) Etant donnée une fonction `i_f` d'interprétation des symboles de constante, il faut d'une part s'assurer que ce symbole n'est pas déjà interprété par un objet présent sur la grille (dans ce cas une exception pourra être levée), d'autre part ajouter physiquement cet objet sur la grille et enfin modifier la fonction `i_f` pour prendre en compte la modification. Définir une fonction `add_cste` de type

   ```
         objet array array -> 'a -> int -> int -> forme -> taille
   -> ('a -> 'b list -> valeur)
   -> 'a -> 'c list -> valeur
   ```

   qui permet d'effectuer cette opération.
   (b) Définir une fonction `sup_cste` de type :

   ```
         'a case array array -> 'b -> int -> int
   -> ('b -> 'c list -> valeur)
   -> 'b -> 'd list -> valeur
   ```

   qui permet de supprimer un objet de la grille et modifier en conséquence la fonction d'interprétation des symboles de constante.

2 Formules logiques du premier ordre

Définition de l'ensemble des formules logiques

Etant donnés une signature Σ (i.e., un ensemble de symboles de fonction muni d'une fonction d'arité $ar_\Sigma : \Sigma \to \mathbb{N}$), un ensemble V de variables et un ensemble $\mathbb{P}$ de symboles de prédicat muni d'une fonction d'arité $ar_\mathbb{P} : \mathbb{P} \to \mathbb{N}$, l'ensemble $\mathcal{F}$ des formules de la logique du premier ordre est défini inductivement comme suit :

- Si p est un symbole de prédicat d'arité n et si t_1, ..., t_n sont des termes, alors $p(t_1, \cdots, t_n) \in \mathcal{F}$ (il s'agit d'une formule atomique).
- Si $\varphi \in \mathcal{F}$ alors $\neg\varphi \in \mathcal{F}$.
- Si $\varphi, \psi \in \mathcal{F}$ alors $\varphi \wedge \psi$, $\varphi \vee \psi$ et $\varphi \Rightarrow \psi$ sont des éléments de $\mathcal{F}$.
- Si x est un symbole de variable et si $\varphi \in \mathcal{F}$ alors $\forall x\, \varphi \in \mathcal{F}$ et $\exists x\, \varphi \in \mathcal{F}$.

On définit donc le type :

```
type ('a,'b,'c) form =
     Atom   of 'c * (('a,'b) term) list
  |  Impl   of ('a,'b,'c) form * ('a,'b,'c) form
  |  And    of ('a,'b,'c) form * ('a,'b,'c) form
  |  Or     of ('a,'b,'c) form * ('a,'b,'c) form
  |  Neg    of ('a,'b,'c) form
  |  Forall of 'a * ('a,'b,'c) form
  |  Exists of 'a * ('a,'b,'c) form;;
```

`'a` est le type des variables, `'b` le type des fonctions et `'c` le type des prédicats.

On définit l'ensemble des variables qui admettent au moins une occurrence libre dans une formule φ par :

$$\vartheta_F(\varphi) = \begin{cases} \vartheta(t_1) \cup \cdots \cup \vartheta(t_n) & \text{si } \varphi = p(t_1, \cdots, t_n) \\ \vartheta_F(\psi) & \text{si } \varphi = \neg\psi \\ \vartheta_F(\psi_1) \cup \vartheta_F(\psi_2) & \text{si } \varphi = \psi_1 \vee \psi_2 \text{ ou } \psi_1 \wedge \psi_2 \text{ ou } \psi_1 \Rightarrow \psi_2 \\ \vartheta_F(\psi)\backslash\{x\} & \text{si } \varphi = \forall x\psi \text{ ou } \exists x\psi \end{cases}$$

Exercice 2 Définir une fonction : `free_var_F :('a,'b,'c) form->'a list` qui calcule $\vartheta_F(\varphi)$.

Application d'une substitution à une variable

Substituer une variable x par un terme t dans une formule φ pour obtenir une formule notée $\varphi[x := t]$ consiste à remplacer *certaines* occurrences de x dans φ par le terme t.

Exemple. Les formules $(\forall x(p(x) \wedge q(x,y))) \vee (\exists z q(x,z))$ et $(\forall w(p(w) \wedge q(w,y))) \vee (\exists z q(x,z))$ sont "logiquement équivalentes" ... en substituant x par $f(v)$, on souhaite donc obtenir des formules "logiquement équivalentes" ... ce qui n'est pas le cas si l'on substitue toutes les occurrences de x :

$$\begin{array}{rl} & ((\forall x(p(x) \wedge q(x,y))) \vee (\exists z q(x,z)))[x := f(v)] \\ \stackrel{?}{=} & (\forall x(p(f(v)) \wedge q(f(v),y))) \vee (\exists z q(f(v),z)) \\ \hline & ((\forall w((p(w) \wedge q(w,y)))) \vee (\exists z q(x,z)))[x := f(v)] \\ \stackrel{?}{=} & (\forall w(p(w) \wedge q(w,y))) \vee (\exists z q(f(v),z)) \end{array}$$

C'est pourquoi, l'application de la substitution ne porte que sur les occurrences libres de x :

$$\begin{array}{rl} & ((\forall x(p(x) \wedge q(x,y))) \vee (\exists z q(x,z)))[x := f(v)] \\ = & (\forall x(p(x) \wedge q(x,y))) \vee (\exists z q(f(v),z)) \\ \hline & ((\forall w(p(w) \wedge q(w,y))) \vee (\exists z q(x,z)))[x := f(v)] \\ = & (\forall w(p(w) \wedge q(w,y))) \vee (\exists z q(f(v),z)) \end{array}$$

Toutefois, ceci n'est correct que si les variables apparaissant dans t ne sont pas liées dans φ, dans le cas contraire il y a capture de variable (voir cours). Il faut donc, avant d'effectuer la substitution, renommer les variables liées de φ qui apparaissent dans t.

L'implantation de l'application d'une substitution à une formule est donc définie par la fonction suivante :

```
exception Subst_error;;
let rec subst_form x t f = match f with
   Atom(p,l)   -> Atom(p,(List.map (apply_subst_T [(x,t)]) l))
 | Impl(f1,f2) -> Impl((subst_form x t f1),(subst_form x t f2))
 | And(f1,f2)  -> ...
 | Or(f1,f2)   -> ...
 | Neg(f1)     -> Neg((subst_form x t f1))
 | Forall(y,fy) -> if x=y
                   then fy
                   else if (List.mem y (var_T t))
                        then raise Subst_error
                        else Forall(y,(subst_form x t fy))
 | Exists(y,fy) -> ...
```

Interprétation des formules

Afin de pouvoir interpréter les formules du premier ordre, on enrichit la notion d'algèbre : une $(\Sigma \cup \mathbb{P})$-algèbre est définie par un triplet $(\mathcal{D}, \mathcal{I}_\Sigma, \mathcal{I}_\mathbb{P})$ où

- $\mathcal{D}$ est le domaine d'interprétation
- $\mathcal{I}_\Sigma$ est une interprétation des symboles de fonction qui permet d'associer à tout $f \in \Sigma$ d'arité n une fonction $[\![f]\!]_{\mathcal{I}_\Sigma} : \mathcal{D}^n \rightarrow \mathcal{D}$
- $\mathcal{I}_\mathbb{P}$ est une interprétation des symboles de prédicat qui permet d'associer à tout $p \in \mathbb{P}$ d'arité n une fonction $[\![p]\!]_{\mathcal{I}_\mathbb{P}} : \mathcal{D}^n \rightarrow \mathbb{B}$

Le schéma d'interprétation des formules est alors défini comme suit. Soit $\mathcal{A} = (\mathcal{D}, \mathcal{I}_\Sigma, \mathcal{I}_\mathbb{P})$ une $(\Sigma \cup \mathbb{P})$-algèbre et ν une valuation.

$$
\begin{array}{l}
[\![p(t_1, \cdots, t_n)]\!]_{\mathcal{A},\nu} = [\![p]\!]_{\mathcal{I}_\mathbb{P}}([\![t_1]\!]_{\mathcal{A},\nu}, \cdots, [\![t_n]\!]_{\mathcal{A},\nu}) \\
[\![\neg\psi]\!]_{\mathcal{A},\nu} = [\![\neg]\!][\![\psi]\!]_{\mathcal{A},\nu} \\
[\![\psi_1 \wedge \psi_2]\!]_{\mathcal{A},\nu} = [\![\psi_1]\!]_{\mathcal{A},\nu}[\![\wedge]\!][\![\psi_2]\!]_{\mathcal{A},\nu} \\
[\![\psi_1 \vee \psi_2]\!]_{\mathcal{A},\nu} = [\![\psi_1]\!]_{\mathcal{A},\nu}[\![\vee]\!][\![\psi_2]\!]_{\mathcal{A},\nu} \\
[\![\psi_1 \Rightarrow \psi_2]\!]_{\mathcal{A},\nu} = [\![\psi_1]\!]_{\mathcal{A},\nu}[\![\Rightarrow]\!][\![\psi_2]\!]_{\mathcal{A},\nu} \\
[\![\forall x \psi]\!]_{\mathcal{A},\nu} = \mathsf{true} \text{ ssi pour tout } a \in \mathcal{D}\ [\![\psi[x := a]]\!]_{\mathcal{A},\nu} = \mathsf{true} \\
[\![\exists x \psi]\!]_{\mathcal{A},\nu} = \mathsf{true} \text{ ssi il existe } a \in \mathcal{D} \text{ tq } [\![\psi[x := a]]\!]_{\mathcal{A},\nu} = \mathsf{true}
\end{array}
$$

L'implantation de ce schéma est obtenue avec la fonction :

```
let rec interp_form v i_f i_p check_forall check_exists f = match f with
   Atom(p,l)   -> ((i_p p) (List.map (interp_term v i_f) l))
 | Impl(f1,f2) -> (not
                    (interp_form v i_f i_p check_forall check_exists f1))
                   or (interp_form v i_f i_p check_forall check_exists f2)
 | And(f1,f2)  -> (interp_form v i_f i_p check_forall check_exists f1) &&
                  (interp_form v i_f i_p check_forall check_exists f2)
 | Or(f1,f2)   -> (interp_form v i_f i_p check_forall check_exists f1) or
                  (interp_form v i_f i_p check_forall check_exists f2)
 | Neg(f1)     -> not
                   (interp_form v i_f i_p check_forall check_exists f1)
 | Forall(x,fx) -> (check_forall x v i_f i_p fx)
 | Exists(x,fx) -> (check_exists x v i_f i_p fx);;
```

dont le type est :

```
    ('a -> 'b)                                    /*1*/
->  ('c -> 'b list -> 'b)                         /*2*/
->  ('d -> 'b list -> bool)                       /*3*/
->  ('a -> ('a -> 'b) -> ('c -> 'b list -> 'b) -> /*4*/
    ('d -> 'b list -> bool) -> ('a, 'c, 'd) form -> bool)
->  ('a -> ('a -> 'b) -> ('c -> 'b list -> 'b) -> /*5*/
    ('d -> 'b list -> bool) -> ('a, 'c, 'd) form -> bool)
->  ('a, 'c, 'd) form
-> bool
```

où `'a` désigne le type des variables, `'b` le type des éléments du domaine d'interprétation, `'c` le type des symboles de fonction, et `'d` le type des symboles de prédicat. Aussi, le premier argument (`/*1*/`) correspond à une valuation, le deuxième argument (`/*2*/`) correspond à une interprétation des symboles de fonction, le troisième argument (`/*3*/`) correspond à une interprétation des symboles de prédicat, le quatrième argument (`/*4*/`) correspond à une fonction qui étant donnée une variable x, une valuation v, une interprétation des symboles de fonction, une interprétation des symboles de prédicat, et une formule φ permet d'indiquer si pour tout élément a appartenant au domaine d'interprétation, la formule $\varphi[x := a]$ est interprétée à `true`, et enfin le dernier argument (`/*5*/`) est une fonction similaire au quatrième argument et qui permet d'indiquer si il existe un élément a appartenant au domaine d'interprétation tel que la formule $\varphi[x := a]$ est interprétée à `true`. Les deux derniers arguments correspondent donc à des fonctions permettant de parcourir le domaine d'interprétation (puisque ce domaine est arbitraire, ces deux fonctions sont passées en argument). Si ce domaine n'est pas fini, ces fonctions ne terminent pas nécessairement.

Formules de Tarski's world

Les formules logiques de Tarski's world sont construites à partir de :

- V : ensemble de symboles de variable
  ```
  type variables = VAR of int;;
  ```
- Σ : ensemble de symboles de fonction
  ```
  type constantes =
      A | B | C | D | E | F | G | H | I | J | K | L | M |
      N | O | P | Q | R | S | T | U | V | W | X | Y | Z;;
  ```
- $\mathbb{P}$: ensemble de symboles de prédicat
  ```
  type predicats =
      Tet     | Cub    | Dodec  | Small   | Medium | Large
    | Smaller | Larger | LeftOf | RightOf | BackOf | FrontOf
    | Between;;
  ```

Placer les objets sur la grille, c'est construire une interprétation ...

- obtention d'un monde de Tarski :
  ```
  w : objet case array array
  ```
- obtention d'une interprétation des constantes :
  ```
  i_f : constantes -> valeur list -> valeur
  ```
  `i_f` fournit les coordonnées dans la grille `w` des objets désignés par une constante.

Il reste à donner l'interprétation des symboles de $\mathbb{P}$:

```
let interp_pred w =
  function p -> match p with
                  Cub -> (interp_Cub w)
                | ... ;;
(* (interp_pred w) : predicats -> valeur list -> bool *)
```

Par exemple, on pourra définir `interp_Cub` par :

```
let interp_Cub w l = match l with
    [x] -> (match x with
                Obj(i,j) -> (match (w.(i).(j)) with
                                 Case o ->  o.form = Cube
                               | Empty -> failwith "error")
              | Undef -> failwith "undefined obj")
  | _ -> failwith "arity_error";;

(* interp_Cub : objet case array array -> valeur list -> bool *)
```

Les fonctions `check_forall` et `check_exists`, arguments de `interp_form`, permettent de parcourir le domaine ... elles sont mutuellement récursives puisqu'elles appellent la fonction `interp_form`.

```
let rec interp_form_tarski w v i_f f =
  (interp_form v i_f (interp_pred w) (check_forall w) (check_exists w) f)
and check_forall w x v i_f i_p fx =
 (List.for_all (interp_form_tarski w v i_f)
                (List.map (function t -> subst_form x (Cons_term(t,[])) fx)
                  (List.filter (function x -> (match (i_f x []) with
                                      Obj(i,j) -> true
                                     |Undef -> false))
                        [A;B;C;D;E;...;Z])))
and check_exists w x v i_f i_p fx =
 (List.exists ...
```

Exercice 3

1. Définir une fonction :

   ```
   interp_pred : objet case array array predicats -> valeur list -> bool
   ```

   qui permet d'interpréter les symboles de prédicat.

2. Définir trois fonctions :

   ```
   interp_form_tarski :
         objet case array array
     ->  ('a -> valeur)
     ->  (constantes -> valeur list -> valeur)
     ->  ('a, constantes, predicats) form
     ->  bool

   check_forall :
         objet case array array
     ->  'a
     ->  ('a -> valeur)
     ->  (constantes -> valeur list -> valeur)
     ->  (predicats -> valeur list -> bool)
     ->  ('a, constantes, predicats) form
     ->  bool

   check_exists :
         objet case array array
     ->  'a
     ->  ('a -> valeur)
     ->  (constantes -> valeur list -> valeur)
     ->  (predicats -> valeur list -> bool)
     ->  ('a, constantes, predicats) form
     ->  bool
   ```

   qui permettent d'interpréter les formules logiques de Tarski's world.

Fonctions prédéfinies utilisées

• `val assoc : 'a -> ('a * 'b) list -> 'b`
`assoc a l` *returns the value associated with key* `a` *in the list of pairs* `l`. *That is,*

```
assoc a [ ...; (a,b); ...] = b
```

if `(a,b)` *is the leftmost binding of* `a` *in list* `l`. *Raise* `Not_found` *if there is no value associated with* `a` *in the list* `l`.

```
let rec assoc c l = match l with
   [] -> raise Not_found
 | (k,v)::t -> if c=k then v else (assoc c t);;
```

• `val exists : ('a -> bool) -> 'a list -> bool`
`exists p [a1; ...; an]` *checks if at least one element of the list satisfies the predicate* `p`. *That is, it returns* `(p a1) || (p a2) || ... || (p an)`.

$$(\texttt{exists}\ p\ l) = \texttt{true} \Leftrightarrow \exists x \in l\ p(x)$$

```
let rec exists p l = match l with
    [] -> false
  | h::t -> if (p h) then true else (exists p t);;

# List.exists (function x -> (x mod 2)=0) [1;3];;
- : bool = false
# List.exists (function x -> x>0) [1;3];;
- : bool = true
```

• `val filter : ('a -> bool) -> 'a list -> 'a list`
`filter p l` *returns all the elements of the list* `l` *that satisfy the predicate* `p`. *The order of the elements in the input list is preserved.*

```
let rec filter p l = match l with
    [] -> []
  | h::t -> if (p h) then h::(filter p t) else (filter p t);;

# List.filter (function x -> (x mod 2)=0) [1;2;3;4];;
- : int list = [2; 4]
```

• `val for_all : ('a -> bool) -> 'a list -> bool`
`exists p [a1; ...; an]` *checks if all elements of the list satisfies the predicate* `p`. *That is, it returns* `(p a1) && (p a2) && ... && (p an)`.

$$(\texttt{for\_all}\ p\ l) = \texttt{true} \Leftrightarrow \forall x \in l\ p(x)$$

```
let rec forall p l = match l with
    [] -> true
  | h::t -> if (p h) then (for_all p t) else false;;

# List.for_all (function x -> (x mod 2)=0) [1;3];;
- : bool = false
# List.for_all (function x -> x>0) [1;3];;
- : bool = true
```

• `val map : ('a -> 'b) -> 'a list -> 'b list`
`map f [a1;..;an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

```
let rec map f l = match l with
       [] -> []
     | (a::r) -> (f a) :: (map f r);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# int_of_char;;
- : char -> int = <fun>
# List.map int_of_char ['a';'z';'A';'Z'];;
- : int list = [97; 122; 65; 90]
```

• `val mem : 'a -> 'a list -> bool`
`mem a l` *is true if and only if* `a` *is equal to an element of* `l`.

```
let rec mem a l = match l with
   [] -> false
 | h::t -> if a=h then true else (mem a t);;
```