Interprétation d'une formule de la logique des prédicats

1 Termes

1.1 Syntaxe

1.1.1 Définition et implantation

L'ensemble des termes est construit à partir d'un ensemble $\mathcal{F}$ de symboles de fonction et d'un ensemble X (disjoint de $\mathcal{F}$) de symboles de variable. L'ensemble $\mathcal{F}$ est appelé une signature et est muni d'une application $ar : \mathcal{F} \to \mathbb{N}$ spécifiant l'arité des symboles de fonction (l'arité d'un symbole de fonction désigne le nombre d'arguments de cette fonction). On note $\mathcal{F}_n$ le sous-ensemble de $\mathcal{F}$ contenant les symboles de fonction d'arité n. Ainsi, $\mathcal{F}$ peut s'écrire $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \cdots = \bigcup_{n \geq 0} \mathcal{F}_n$. Les symboles appartenant à $\mathcal{F}_0$ correspondent aux symboles de constante (ils n'ont pas d'argument). On note souvent a, b, c, ... les constantes, f, g, h, ... les symboles de fonction d'arité strictement positive, et v, w, x, y, z les symboles de variable. L'ensemble $T_{\mathcal{F}}[X]$ des termes est défini inductivement comme suit.

1. Tout symbole de variable est un terme ($X \subseteq T_{\mathcal{F}}[X]$).
2. Si $f \in \mathcal{F}_n$ 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.

En particulier, si $k \in \mathcal{F}_0$ est un symbole de constante, alors k est un terme ($\mathcal{F}_0 \subseteq T_{\mathcal{F}}[X]$). Par exemple, si l'on considère une signature $\mathcal{F}$ contenant les symboles $a \in \mathcal{F}_0$, $b \in \mathcal{F}_0$, $f \in \mathcal{F}_2$ et $g \in \mathcal{F}_3$, et un ensemble de variables X contenant les symboles x et y, alors on peut définir le terme suivant :

$$f(g(x, b, a), g(f(a, y), b, x)) \tag{1}$$

L'ensemble $\vartheta(t)$ des variables d'un terme t est défini récursivement comme suit.

$$\forall t \in T_{\mathcal{F}}[X] \quad \vartheta(t) = \begin{cases} \{x\} & \text{si } t = x \in X \\ \vartheta(t_1) \cup \cdots \cup \vartheta(t_n) & \text{si } t = f(t_1, \cdots, t_n) \end{cases}$$

Par exemple, pour le terme t défini en (1), on a $\vartheta(t) = \{x, y\}$.

Python

Une manière simple d'implanter la notion de terme avec le langage Python consiste à associer une classe à chaque règle de construction des termes. Ici, la notion d'arité des symboles de fonction n'est pas implantée et aucun contrôle n'est donc effectué sur le respect de l'arité des symboles de fonction. Les termes de la forme $f(t_1, \cdots, t_n)$ sont implantés à partir du symbole de fonction f et de la liste de termes $[t_1, \cdots, t_n]$.

```
class Var_term:
    def __init__(self,symbv):
        self.symbv = symbv
class Cons_term:
    def __init__(self,symbf,lsubterms):
        self.symbf = symbf
        self.lsubterms = lsubterms
```

Par exemple, le terme défini en (1) s'écrit :

```
a_term = \
       Cons_term("f",\
                  [Cons_term("g",[Var_term("x"),Cons_term("b",[]),\
                                 Cons_term("a",[])]),\
                   Cons_term("g",[Cons_term("f",[Cons_term("a",[]),\
                                                Var_term("y")]),\
                                 Cons_term("b",[]),Var_term("x")])])
```

Le calcul de $\vartheta(t)$ s'obtient récursivement comme suit.

```
def var_T(eqX,t):
    if isinstance(t,Var_term):
        return singleton(t.symbv)
    if isinstance(t,Cons_term):
        r = emptyset
        for x in t.lsubterms:
            r = union(eqX,r,var_T(eqX,x))
        return r
    raise ValueError
>>> var_T(eq_atom,a_term)
['y', 'x']
```

La fonction `eq_atom` utilisée dans cet exemple est définie comme l'égalité atomique `==` de Python :

```
def eq_atom(x,y):
    return x==y
```

OCaml

Le type des termes peut être défini par le type somme suivant :

```
# 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 $\mathcal{F}$. Ici, la notion d'arité des symboles de fonction n'est pas implantée et aucun contrôle n'est donc effectué sur le respect de l'arité des symboles de fonction. Les termes de la forme $f(t_1, \cdots, t_n)$ sont implantés à partir du symbole de fonction f et de la liste de termes $[t_1; \cdots; t_n]$. Par exemple, le terme défini en (1) s'écrit :

```
# let a_term = Cons_term("f",
    [Cons_term("g",[Var_term("x");Cons_term("b",[]);Cons_term("a",[])]);
     Cons_term("g",[Cons_term("f",[Cons_term("a",[]);Var_term("y")]);Cons_term("b",[]);Var_term("x")])]);;
val a_term : (string, string) term
```

Le calcul de $\vartheta(t)$ s'obtient récursivement comme suit.

```
# let rec var_T eqX t = match t with
 Var_term x -> (singleton x)
|Cons_term(f,l) -> (List.fold_left (function x -> function y -> (union eqX x (var_T eqX y))) emptyset l);;
val var_T : ('a -> 'a -> bool) -> ('a, 'b) term -> 'a list = <fun>
# (var_T eq_atom a_term);;
- : string list = ["y"; "x"]
```

La fonction `eq_atom` utilisée dans cet exemple est définie comme l'égalité atomique `=` de Ocaml :

```
# let eq_atom x y = (x=y);;
val eq_atom : 'a -> 'a -> bool = <fun>
```

1.1.2 Syntaxe des termes du projet

Les termes considérés dans le projet sont des constantes ou des variables (la signature utilisée ne contient aucun symbole de fonction d'arité strictement positive) désignant des fleurs. Ces termes sont construits à partir d'un ensemble fini $\mathcal{F}_0$ contenant 20 constantes et d'un ensemble fini X contenant 5 symboles de variable.

Python

Chaque constante est représentée par un caractère : `a`, `b`, ..., `t` et chaque symbole de variable est représenté par un caractère : `v`, `w`, `x`, `y`, `z`). Un terme correspondant à une constante est une instance de la classe `Cons_term`. Par exemple, le terme obtenu à partir de la constante `a` s'écrit en Python :

```
>>> Cons_term("a",[])
```

Un terme correspondant à un symbole de variable est une instance de la classe `Var_term`. Par exemple, le terme obtenu à partir du symbole de variable `x` s'écrit en Python :

```
>>> Var_term("x")
```

OCaml

On définit les types sommes suivants pour représenter $\mathcal{F}_0$ et X :

```
# type sig_f0 = F0_A|F0_B|F0_C|F0_D|F0_E|F0_F|F0_G|F0_H|F0_I|F0_J|F0_K
               |F0_L|F0_M|F0_N|F0_O|F0_P|F0_Q|F0_R|F0_S|F0_T;;
# type var_x = X_v | X_w | X_x | X_y | X_z ;;
```

Un terme correspondant à une constante est obtenu à partir du constructeur `Cons_term`. Par exemple, le terme obtenu à partir de la constante `F0_A` s'écrit en OCaml :

```
# Cons_term(F0_A,[]);;
- : ('a, sig_f0) term = Cons_term (F0_A, [])
```

Un terme correspondant à un symbole de variable est obtenu à partir du constructeur `Var_term`. Par exemple, le terme obtenu à partir du symbole de variable `X_x` s'écrit en OCaml :

```
# Var_term(X_x);;
- : (var_x, 'a) term = Var_term X_x
```

1.2 Sémantique : Interprétation des termes

1.2.1 Définition et implantation

Interpréter un terme, c'est lui associer une valeur appartenant à un certain ensemble, appelé domaine d'interprétation. Les symboles de variable sont interprétés grâce à la notion de valuation : une valuation est une application de l'ensemble X des symboles de variable dans le domaine d'interprétation. Une valuation permet donc de connaître la valeur associée à chaque variable. La notion de structure permet d'interpréter les symboles de $\mathcal{F}$. Une structure $\mathbf{M}$ est la donnée :

— d'un ensemble non vide $|\mathbf{M}|$, appelé le domaine d'interprétation de $\mathbf{M}$,

— d'une application qui associe à tout symbole $f \in \mathcal{F}$ d'arité n une application $f^{\mathbf{M}} : |\mathbf{M}|^n \to |\mathbf{M}|$.

Etant données une structure $\mathbf{M}$ et une valuation $\nu : X \to |\mathbf{M}|$, la fonction d'interprétation des termes $[\![\,]\!]_\nu^{\mathbf{M}} : T_{\mathcal{F}}[X] \to |\mathbf{M}|$ est définie par :

$$[\![t]\!]_\nu^{\mathbf{M}} = \begin{cases} \nu(x) & \text{si } t = x \in X \\ f^{\mathbf{M}}([\![t_1]\!]_\nu^{\mathbf{M}}, \cdots, [\![t_n]\!]_\nu^{\mathbf{M}}) & \text{si } t = f(t_1, \cdots, t_n) \end{cases}$$

Python

Le schéma d'interprétation des termes est défini à partir d'une valuation `v`, et de l'interprétation `interp_f` des symboles de la signature (chaque fonction d'arité n est implantée par une fonction dont l'argument est une liste de longueur n contenant les arguments de f).

```
def interp_term(v,interp_f,t):
    if isinstance(t,Var_term):
        return v(t.symbv)
    if isinstance(t,Cons_term):
        f = interp_f(t.symbf)
        l1 = t.lsubterms
        l2 = []
        for i in range(0,len(l1)):
            l2 = l2 + [interp_term(v,interp_f,l1[i])]
        return f(l2)
    raise ValueError
```

OCaml

Le schéma d'interprétation des termes est défini à partir d'une valuation `v`, et de l'interprétation `interp_f` des symboles de la signature (chaque fonction d'arité n est implantée par une fonction dont l'argument est une liste de longueur n contenant les arguments de f).

```
# 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));;
val interp_term : ('a -> 'b) -> ('c -> 'b list -> 'b) -> ('a, 'c) term -> 'b = <fun>
```

1.2.2 Interprétation des termes du projet

Construction d'un jardin Les termes considérés dans le projet désignent des fleurs positionnées sur une grille. La figure 1 représente la grille : les points noirs symbolisent les places où peuvent être disposées les fleurs (il y a au plus une fleur par place). Chaque place est identifiée par ses coordonnées (le point (0,0) est au centre de la grille). L'ensemble des places possibles est donc :

$$\text{Places} = \left\{ \begin{array}{c} (0,7), \\ (-1,6),(1,6), \\ (-2,5),(0,5),(2,5), \\ (-3,4),(-1,4),(1,4),(3,4), \\ (-4,3),(-2,3),(0,3),(2,3),(4,3), \\ (-5,2),(-3,2),(-1,2),(1,2),(3,2),(5,2), \\ (-6,1),(-4,1),(-2,1),(0,1),(2,1),(4,1),(6,1), \\ (-7,0),(-5,0),(-3,0),(-1,0),(1,0),(3,0),(5,0),(7,0), \\ (-6,-1),(-4,-1),(-2,-1),(0,-1),(2,-1),(4,-1),(6,-1), \\ (-5,-2),(-3,-2),(-1,-2),(1,-2),(3,-2),(5,-2), \\ (-4,-3),(-2,-3),(0,-3),(2,-3),(4,-3), \\ (-3,-4),(-1,-4),(1,-4),(3,-4), \\ (-2,-5),(0,-5),(2,-5), \\ (-1,-6),(1,-6), \\ (0,-7) \end{array} \right\}$$

Chaque fleur appartient à une espèce (les roses, les pâquerettes et les tulipes), est d'une certaine taille (grande, moyenne ou petite) et d'une certaine couleur (rouge, rose, blanche).

$$\text{Especes} = \{\text{rose}, \text{paquerette}, \text{tulipe}\} \quad \text{Tailles} = \{\text{grand}, \text{moyen}, \text{petit}\} \quad \text{Couleurs} = \{\text{rouge}, \text{rose}, \text{blanc}\}$$

Certaines fleurs ont un nom : il s'agit d'une constante de $\mathcal{F}_0$ (deux fleurs différentes ne peuvent pas avoir le même nom). Un jardin j est la donnée d'une grille sur laquelle sont disposées des fleurs (chaque jardin contient au moins une fleur) et est représenté par une liste de quintuplets. Chaque quintuplet $((x,y),e,t,c,n) \in j$ est un élément du produit cartésien :

$$\text{Places} \times \text{Especes} \times \text{Tailles} \times \text{Couleurs} \times (\mathcal{F}_0 \cup \{\texttt{None}\})$$

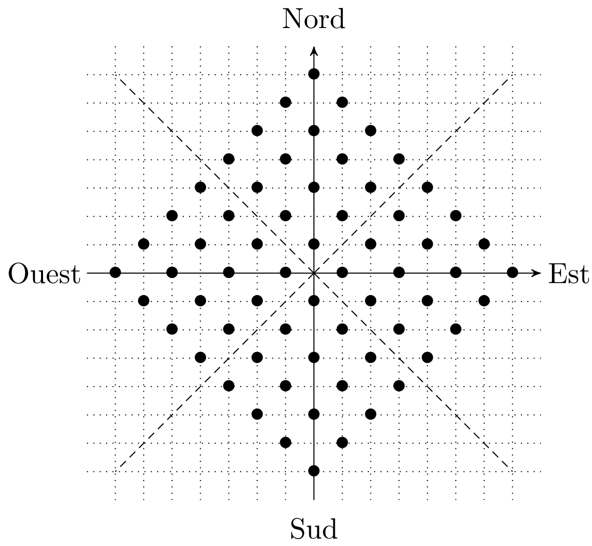

FIGURE 1 – Places d'un jardin

et exprime qu'une fleur de nom n ($n =$ `None` si la fleur n'a pas de nom), d'espèce e, de taille t et de couleur c se trouve à la place (x, y) dans le jardin j. L'ensemble des jardins possibles est donc :

$$J = \wp(\text{Places} \times \text{Especes} \times \text{Tailles} \times \text{Couleurs} \times (\mathcal{F}_0 \cup \{\texttt{None}\}))\backslash\{\emptyset\}$$

Python

On représente ici les noms d'espèce, de taille, et de couleur par des chaînes de caractères. Un jardin est représenté par une liste de quintuplets. On peut par exemple définir le jardin suivant :

```
mon_jardin = [((-3,2),"tulipe","moyen","rose","a"),((2,3),"rose","petit","blanc",None),\
              ((-2,-1),"rose","grand","rouge",None),((2,1),"paquerette","moyen","blanc","d"),\
              ((4,-3),"rose","petit","rose",None),((6,-1),"tulipe","petit","rouge","f"),\
              ((7,0),"paquerette","grand","rouge","g")]
```

OCaml

On représente ici les espèces, les tailles et les couleurs avec les types sommes suivants :

```
# type especes = E_rose | E_paquerette | E_tulipe ;;
# type tailles = T_grand | T_moyen | T_petit ;;
# type couleurs = C_rouge | C_rose | C_blanc ;;
```

Un jardin est représenté par une liste de quintuplets. On peut par exemple définir le jardin suivant :

```
# let mon_jardin =
  [((-3,2),E_tulipe,T_moyen,C_rose,Some(F0_A)) ; ((2,3),E_rose,T_petit,C_blanc,None) ;
   ((-2,-1),E_rose,T_grand,C_rouge,None) ; ((2,1),E_paquerette,T_moyen,C_blanc,Some(F0_D)) ;
   ((4,-3),E_rose,T_petit,C_rose,None)   ; ((6,-1),E_tulipe,T_petit,C_rouge,Some(F0_F)) ;
   ((7,0),E_paquerette,T_grand,C_rouge,Some(F0_G))];;
val mon_jardin : ((int * int) * especes * tailles * couleurs * sig_f0 option) list
```

On utilise le type `option` pour représenter le nom d'une fleur : le constructeur `None` est utilisé pour une fleur sans nom, et le constructeur `Some` appliqué sur une valeur de type `sig_f0` est utilisé pour une fleur qui a un nom.

Construction d'une structure à partir d'un jardin Etant donné un jardin $j \in J$, il est possible de construire une structure $\mathbf{M}_j$ permettant d'interpréter l'ensemble de termes $T_{\mathcal{F}_0}[X]$. La structure $\mathbf{M}_j$ est définie par :

— un domaine d'interprétation contenant toutes les places de j qui sont occupées par une fleur, auquel on ajoute un élément particulier (Error) :

$$|\mathbf{M}_j| = \{(x,y) \in \text{Places} \mid ((x,y),e,t,c,n) \in j\} \cup \{\text{Error}\}$$

— une fonction d'interprétation qui associe un élément de $|\mathbf{M}_j|$ à chaque symbole de constante de $\mathcal{F}_0$ désignant une fleur du jardin j : il s'agit de la place (x,y) de cette fleur dans le jardin (si aucune fleur de nom n se trouve dans le jardin, l'interprétation de n déclenche une erreur qui peut être vue comme une valeur particulière Error du domaine d'interprétation) :

$$n^{\mathbf{M}_j} = \begin{cases} (x,y) & \text{si } ((x,y),e,t,c,n) \in j \\ \text{Error} & \text{sinon} \end{cases}$$

Python

Pour calculer le domaine de la structure associée à un jardin, il suffit de parcourir l'ensemble des places du jardin qui contiennent une fleur (on ne considère pas ici la valeur Error) :

```
def j_domain(j):
    return [(x,y) for ((x,y),e,t,c,n) in j]
>>> j_domain(mon_jardin)
[(-3, 2), (2, 3), (-2, -1), (2, 1), (4, -3), (6, -1), (7, 0)]
```

L'interprétation des symboles de $\mathcal{F}_0$ s'obtient comme suit :

```
def j_interp_f(j):
    def _interp_f0(k):
        def sln():
            for (p,_,_,_,n) in j:
                if n==k:
                    return p
            raise ValueError
        def __interp_k(l):
            if len(l)==0:
                return sln()
            else:
                raise ValueError
        return __interp_k;
    return _interp_f0
```

Etant donné un jardin `j`, `j_interp_f(j)` retourne la fonction `_interp_f0` qui correspond à une fonction qui étant donné un symbole de constante `k` retourne la fonction qui correspond à l'interprétation de ce symbole de constante. Cette interprétation correspond à une fonction d'arité 0, et est donc une fonction qui étant donnée une liste `l` déclenche une erreur si cette liste n'est pas vide, et sinon retourne la place de la fleur de nom `k` dans le jardin ou déclenche une erreur si aucune fleur du jardin ne porte le nom `k`.

```
mon_interp_f = j_interp_f(mon_jardin)
mon_interp_d = mon_interp_f("d")
>>> mon_interp_d([])
(2, 1)
```

En utilisant la fonction `interp_term` définie plus haut, on obtient alors directement :

```
def valuation_error(x):
    raise ValueError
>>> interp_term(valuation_error,j_interp_f(mon_jardin),Cons_term("d",[]))
(2, 1)
```

Sur cet exemple, le terme considéré ne contient pas de variable et on utilise ici une valuation indéfinie qui déclenche une erreur dès qu'elle est appelée.

OCaml

Pour calculer le domaine de la structure associée à un jardin, il suffit de parcourir l'ensemble des places du jardin qui contiennent une fleur (on ne considère pas ici la valeur Error) :

```
# let j_domain l = (List.map (function (x,_,_,_,_) -> x) l);;
val j_domain : ('a * 'b * 'c * 'd * 'e) list -> 'a list = <fun>
# (j_domain mon_jardin);;
- : (int * int) list =
[(-3, 2); (2, 3); (-2, -1); (2, 1); (4, -3); (6, -1); (7, 0)]
```

L'interprétation des symboles de $\mathcal{F}_0$ s'obtient comme suit :

```
# let j_interp_f j k =
 let rec sln sj = match sj with
     [] -> failwith("ValueError")
  | (p,_,_,_,(Some(n)))::r -> if n=k then p else (sln r)
  | (p,_,_,_,None)::r -> (sln r)
 in function l -> match l with  [] -> (sln j)
                              | _ -> failwith("ValueError");;
val j_interp_f : ('a * 'b * 'c * 'd * 'e option) list -> 'e -> 'f list -> 'a = <fun>
```

En utilisant le mécanisme d'application partielle, étant donné un jardin `j`, `(j_interp_f j)` est une fonction qui étant donné un symbole de constante `k` retourne la fonction qui correspond à l'interprétation de ce symbole de constante. Cette interprétation correspond à une fonction d'arité 0, et est donc une fonction qui étant donnée une liste `l` déclenche une erreur si cette liste n'est pas vide, et sinon retourne la place de la fleur de nom `k` dans le jardin ou déclenche une erreur si aucune fleur du jardin ne porte le nom `k`.

```
# let mon_interp_f = (j_interp_f mon_jardin);;
val mon_interp_f : sig_f0 -> '_a list -> int * int = <fun>
# let mon_interp_d = (mon_interp_f F0_D);;
val mon_interp_d : '_a list -> int * int = <fun>
# (mon_interp_d []);;
- : int * int = (2, 1)
```

En utilisant la fonction `interp_term` définie plus haut, on obtient alors directement :

```
# let valuation_error x = failwith("ValueError");;
val valuation_error : 'a -> 'b = <fun>
# (interp_term valuation_error (j_interp_f mon_jardin) (Cons_term(F0_D,[])));;
- : int * int = (2, 1)
```

Sur cet exemple, le terme considéré ne contient pas de variable et on utilise ici une valuation indéfinie qui déclenche une erreur dès qu'elle est appelée.

2 Formules de la logique des prédicats

2.1 Syntaxe

2.1.1 Définition et implantation

L'ensemble $\mathbf{F}_1$ des formules de la logique des prédicats est défini à partir d'une signature $\mathcal{F}$, d'un ensemble de variables X et d'un ensemble $\mathcal{P}$ de symboles de prédicat. Un symbole de prédicat correspond au nom d'une propriété portant sur des termes de l'ensemble $T_{\mathcal{F}}[X]$. Comme l'ensemble $\mathcal{F}$, l'ensemble $\mathcal{P}$ est muni d'une application $ar : \mathcal{P} \to \mathbb{N}$ permettant d'associer à tout symbole $p \in \mathcal{P}$ son arité $ar(p)$ qui désigne le nombre de termes sur lesquels porte la propriété exprimée par p. On note $\mathcal{P}_n$ le sous-ensemble de $\mathcal{P}$ contenant les symboles de prédicat d'arité n. Ainsi l'ensemble $\mathcal{P}$ peut s'écrire $\mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup \cdots = \bigcup_{n \geq 0} \mathcal{P}_n$. L'ensemble $\mathbf{F}_1$ des formules de la logique des prédicats est défini inductivement comme suit.

1. $true \in \mathbf{F}_1$ et $false \in \mathbf{F}_1$
2. Si p est un symbole d'arité n de $\mathcal{P}_n$, et si t_1, ...,t_n sont des termes dans $T_{\mathcal{F}}[X]$, alors $p(t_1, \cdots, t_n)$ est une formule atomique de $\mathbf{F}_1$.
3. Si $\varphi \in \mathbf{F}_1$ alors $\neg\varphi \in \mathbf{F}_1$.
4. Si $\varphi, \psi \in \mathbf{F}_1$ alors $\varphi \wedge \psi \in \mathbf{F}_1$, $\varphi \vee \psi \in \mathbf{F}_1$, et $\varphi \Rightarrow \psi \in \mathbf{F}_1$.
5. Si x est une variable de X et si $\varphi \in \mathbf{F}_1$ alors $\forall x\ \varphi \in \mathbf{F}_1$ et $\exists x\ \varphi \in \mathbf{F}_1$.

Python

Pour implanter les formules, nous définissons une classe par constructeur de formule. Les formules atomiques de la forme $p(t_1, \cdots, t_n)$ sont implantées à partir du symbole de prédicat p et de la liste de termes $[t_1, \cdots, t_n]$ mais aucun contrôle sur l'arité de p n'est effectué.

```
class F1_true:
    pass
class F1_false:
    pass
class F1_Atom:
    def __init__(self,symbpred,lterms):
        self.symbpred = symbpred
        self.lterms = lterms
class F1_Neg:
    def __init__(self,sub_f):
        self.sub_f = sub_f
class F1_And:
    def __init__(self,sub_f1,sub_f2):
        self.sub_f1 = sub_f1
        self.sub_f2 = sub_f2
class F1_Or:
    def __init__(self,sub_f1,sub_f2):
        self.sub_f1 = sub_f1
        self.sub_f2 = sub_f2
class F1_Impl:
    def __init__(self,sub_f1,sub_f2):
        self.sub_f1 = sub_f1
        self.sub_f2 = sub_f2
class F1_Forall:
    def __init__(self,symbvar,sub_f):
        self.symbvar = symbvar
        self.sub_f = sub_f
class F1_Exists:
    def __init__(self,symbvar,sub_f):
        self.symbvar = symbvar
        self.sub_f = sub_f
```

Par exemple, la formule $\forall x\ \forall y\ (p(x, y) \Rightarrow p(y, x))$ s'écrit :

```
ex_f1 = F1_Forall("x",F1_Forall("y",\
                                F1_Impl(F1_Atom("p",[Var_term("x"),Var_term("y")]),\
                                        F1_Atom("p",[Var_term("y"),Var_term("x")]))))
```

OCaml

Pour implanter les formules, nous définissons un type somme récursif. Les formules atomiques de la forme $p(t_1, \cdots, t_n)$ sont implantées à partir du symbole de prédicat p et de la liste de termes $[t_1; \cdots; t_n]$ mais aucun contrôle sur l'arité de p n'est effectué.

```
# type ('a,'b,'c) formul1 =  F1_true | F1_false
```

```
 | F1_Atom of 'c * (('a,'b) term) list | F1_Neg  of ('a,'b,'c) formul1
 | F1_And  of ('a,'b,'c) formul1 * ('a,'b,'c) formul1
 | F1_Or   of ('a,'b,'c) formul1 * ('a,'b,'c) formul1
 | F1_Impl of ('a,'b,'c) formul1 * ('a,'b,'c) formul1
 | F1_Forall of 'a * ('a,'b,'c) formul1 | F1_Exists of 'a * ('a,'b,'c) formul1;;
```

Il s'agit d'un type polymorphe où `'a` correspond au type des variables et `'b` au type des symboles de fonction, et `'c` au type des symboles de prédicat. Par exemple, la formule $\forall x\ \forall y\ (p(x,y) \Rightarrow p(y,x))$ s'écrit :

```
# let ex_f1 =
    F1_Forall("x",F1_Forall("y",
                          F1_Impl(F1_Atom("p",[Var_term("x");Var_term("y")]),
                                  F1_Atom("p",[Var_term("y");Var_term("x")]))));;
val ex_f1 : (string, 'a, string) formul1
```

Variables libres Lorsqu'une occurrence de variable dans une formule logique n'est pas dans la portée d'un quantificateur, elle est libre. Déterminer la valeur de vérité d'une formule contenant des occurrences libres de variable nécessite de disposer d'une information ≪ extérieure ≫. Par exemple, pour déterminer la valeur de vérité de la formule atomique $p(x)$ il faut disposer d'information sur x. Ici le nom de la variable x est discriminant et les formules $p(x)$ et $p(y)$ ne sont pas équivalentes puisque x et y peuvent dénoter des objets différents ne vérifiant pas les mêmes propriétés. Lorsqu'une occurrence de variable dans une formule logique est dans la portée d'un quantificateur, elle est liée (par ce quantificateur). Les variables liées d'une formule sont parfois appelées variables muettes car on peut les renommer avec de nouveaux symboles de variable sans changer la propriété exprimée par la formule. Par exemple, les formules $\forall x\ p(x)$ et $\forall y\ p(y)$ ont la même signification qui ne dépend pas du choix du nom de la variable quantifiée : cette variable est liée par le quantificateur $\forall$. Pour déterminer la valeur de vérité de ces formules, il suffit de déterminer si tous les éléments de l'univers du discours (du domaine d'interprétation) vérifient la propriété (aucune information ≪ extérieure ≫ sur ces variables n'est nécessaire). Etant donnée une formule φ, on définit l'ensemble $\mathrm{Free}(\varphi)$ des variables qui admettent au moins une occurrence libre dans φ.

$$\mathrm{Free}(\varphi) = \begin{cases} \emptyset & \text{si } \varphi = true \text{ ou } \varphi = false \\ \bigcup_{i=1}^{n} \vartheta(t_i) & \text{si } \varphi = p(t_1, \cdots, t_n) \\ \mathrm{Free}(\varphi') & \text{si } \varphi = \neg\varphi' \\ \mathrm{Free}(\varphi_1) \cup \mathrm{Free}(\varphi_2) & \text{si } \varphi = \varphi_1 \wedge \varphi_2 \text{ ou } \varphi = \varphi_1 \vee \varphi_2 \\ & \quad \text{ou } \varphi = \varphi_1 \Rightarrow \varphi_2 \\ \mathrm{Free}(\varphi') \backslash \{x\} & \text{si } \varphi = \forall x\ \varphi' \text{ ou } \varphi = \exists x\ \varphi' \end{cases}$$

Lorsque $\mathrm{Free}(\varphi) = \emptyset$, on dit que la formule φ est close.

Python

Le calcul de $\mathrm{Free}(\varphi)$ s'obtient récursivement comme suit.

```
def free(eqX,f):
    def _varT(t):
        return var_T(eqX,t)
    if isinstance(f,F1_true):
        return emptyset
    if isinstance(f,F1_false):
        return emptyset
    if isinstance(f,F1_Atom):
        return union_sets(eqX,[_varT(t) for t in f.lterms])
    if isinstance(f,F1_Neg):
        return free(eqX,f.sub_f)
    if isinstance(f,F1_And):
```

```
        return union(eqX,free(eqX,f.sub_f1),free(eqX,f.sub_f2))
    if isinstance(f,F1_Or):
        return union(eqX,free(eqX,f.sub_f1),free(eqX,f.sub_f2))
    if isinstance(f,F1_Impl):
        return union(eqX,free(eqX,f.sub_f1),free(eqX,f.sub_f2))
    if isinstance(f,F1_Forall):
        return diff_set(eqX,free(eqX,f.sub_f),singleton(f.symbvar))
    if isinstance(f,F1_Exists):
        return diff_set(eqX,free(eqX,f.sub_f),singleton(f.symbvar))
    raise ValueError
>>> free(eq_atom,ex_f1)
[]
```

La fonction suivante permet de déterminer si une formule est close.

```
def ground(eqX,f):
    return free(eqX,f) == emptyset
>>> ground(eq_atom,ex_f1)
True
```

OCaml

Le calcul de Free(φ) s'obtient récursivement comme suit.

```
# let rec free eqX f = match f with
   F1_true -> emptyset | F1_false -> emptyset
 | F1_Atom(p,l) -> (union_sets eqX (List.map (var_T eqX) l))
 | F1_Neg(f0) -> (free eqX f0)
 | F1_And(f1,f2) -> (union eqX (free eqX f1) (free eqX f2))
 | F1_Or(f1,f2) -> (union eqX (free eqX f1) (free eqX f2))
 | F1_Impl(f1,f2) -> (union eqX (free eqX f1) (free eqX f2))
 | F1_Forall(x,f0) -> (diff_set eqX (free eqX f0) (singleton x))
 | F1_Exists(x,f0) -> (diff_set eqX (free eqX f0) (singleton x));;
val free : ('a -> 'a -> bool) -> ('a, 'b, 'c) formul1 -> 'a list = <fun>
```

La fonction suivante permet de déterminer si une formule est close.

```
# let ground eqX f = (free eqX f) = emptyset;;
val ground : ('a -> 'a -> bool) -> ('a, 'b, 'c) formul1 -> bool = <fun>
# (ground eq_atom ex_f1);;
- : bool = true
```

2.1.2 Syntaxe des formules du projet

Les formules de logique des prédicats considérées dans le projet sont celles obtenues à partir des ensembles $\mathcal{F}_0$ et X introduits dans la section 1.1.2 et de l'ensemble de symboles de prédicats $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$ où :

$$\begin{array}{l}
\mathcal{P}_1 = \left\{ \begin{array}{l} \text{Rose, Paquerette, Tulipe, est_grand, est_moyen, est_petit,} \\ \text{est_rouge, est_rose, est_blanc, a_l_est, a_l_ouest, au_sud, au_nord} \end{array} \right\} \\
\mathcal{P}_2 = \left\{ \begin{array}{l} \text{a_l_est_de, a_l_ouest_de, au_sud_de, au_nord_de, meme_latitude, meme_longitude,} \\ \text{plus_grand_que, plus_petit_que, meme_taille_que, meme_couleur_que, } \doteq \end{array} \right\} \\
\mathcal{P}_3 = \{\text{est_entre}\}
\end{array}$$

On peut par exemple écrire les huit formules ci-dessous.

(`f1`) `d` est une rose : Rose(`d`)

(`f2`) Toutes les fleurs sont des roses : $\forall x\, \text{Rose}(x)$

(`f3`) Il existe une rose : $\exists x\, \text{Rose}(x)$

(f4) Toute fleur blanche est plus petite qu'une fleur située à son est :
$\forall x\,(\text{est\_blanc}(x) \Rightarrow \exists y\,(\text{plus\_petit\_que}(x,y) \wedge \text{a\_l\_est\_de}(y,x)))$

(f5) Toute fleur est à l'est, ou à l'ouest, ou au sud, ou au nord :
$\forall x\,(\text{a\_l\_est}(x) \vee \text{a\_l\_ouest}(x) \vee \text{au\_sud}(x) \vee \text{au\_nord}(x))$

(f6) Toutes les grandes fleurs sont rouges et il n'existe pas de fleur blanche au sud d'une fleur rouge :
$\forall x\,(\text{est\_grand}(x) \Rightarrow \text{est\_rouge}(x)) \wedge \neg\exists x\,(\text{est\_blanc}(x) \wedge \exists y\,(\text{est\_rouge}(y) \wedge \text{au\_sud\_de}(x,y)))$

(f7) Il existe une fleur rouge au nord de la fleur `g` : $\exists x\,(\text{est\_rouge}(x) \wedge \text{au\_nord\_de}(x,\mathtt{g}))$

(f8) Il existe une unique rose rouge :
$\exists x\ \ ((\text{Rose}(x) \wedge \text{est\_rouge}(x)) \wedge (\forall y\,(\text{Rose}(y) \wedge \text{est\_rouge}(y)) \Rightarrow x \doteq y))$

Python

Les symboles de prédicat sont représentés par des chaînes de caractères. Par exemple, les huit formules ci-dessus s'écrivent :

```
f1 = F1_Atom("Rose",[Cons_term("d",[])])
f2 = F1_Forall("x",F1_Atom("Rose",[Var_term("x")]))
f3 = F1_Exists("x",F1_Atom("Rose",[Var_term("x")]))
f4 = F1_Forall("x",\
              F1_Impl(F1_Atom("est_blanc",[Var_term("x")]),\
                      F1_Exists("y",\
                                F1_And(\
                                    F1_Atom("plus_petit_que",[Var_term("x"),Var_term("y")]),\
                                    F1_Atom("a_l_est_de",[Var_term("y"),Var_term("x")])))))
f5 = F1_Forall("x",\
              F1_Or(F1_Atom("a_l_est",[Var_term("x")]),\
                    F1_Or(F1_Atom("a_l_ouest",[Var_term("x")]),\
                          F1_Or(F1_Atom("au_sud",[Var_term("x")]),\
                                F1_Atom("au_nord",[Var_term("x")])))))
f6 = F1_And(\
    F1_Forall("x",\
             F1_Impl(F1_Atom("est_grand",[Var_term("x")]),\
                     F1_Atom("est_rouge",[Var_term("x")]))),\
    F1_Neg(F1_Exists("x",\
                    F1_And(F1_Atom("est_blanc",[Var_term("x")]),\
                           F1_Exists("y",\
                                     F1_And(\
                                         F1_Atom("est_rouge",[Var_term("x")]),\
                                         F1_Atom("au_sud_de",[Var_term("x"),Var_term("y")])))))))
f7 = F1_Exists("x",\
              F1_And(F1_Atom("est_rouge",[Var_term("x")]),\
                     F1_Atom("au_nord_de",[Var_term("x"),Cons_term("g",[])])))
f8 = F1_Exists("x",\
              F1_And(\
                  F1_And(F1_Atom("Rose",[Var_term("x")]),\
                         F1_Atom("est_rouge",[Var_term("x")])),\
                  F1_Forall("y",\
                            F1_Impl(\
                                F1_And(\
                                    F1_Atom("Rose",[Var_term("y")]),\
                                    F1_Atom("est_rouge",[Var_term("y")])),\
                                F1_Atom("egal",[Var_term("x"),Var_term("y")])))))
```

OCaml

On introduit le type somme suivant pour représenter les symboles de prédicat.

```
# type sig_p = P1_Rose | P1_Paquerette | P1_Tulipe | P1_est_grand | P1_est_moyen | P1_est_petit |
```

```
             P1_est_rouge | P1_est_rose | P1_est_blanc | P1_a_l_est | P1_a_l_ouest | P1_au_sud |
             P1_au_nord | P2_a_l_est_de | P2_a_l_ouest_de | P2_au_sud_de | P2_au_nord_de |
             P2_meme_latitude | P2_meme_longitude | P2_plus_grand_que | P2_plus_petit_que |
             P2_meme_taille_que | P2_meme_couleur_que | P2_egal | P3_est_entre;;
```

Les huit formules ci-dessus s'écrivent alors :

```
let f1 = F1_Atom(P1_Rose,[Cons_term(F0_D,[])]);;
let f2 = F1_Forall("x",F1_Atom(P1_Rose,[Var_term("x")]));;
let f3 = F1_Exists("x",F1_Atom(P1_Rose,[Var_term("x")]));;
let f4 = F1_Forall("x",
             F1_Impl(F1_Atom(P1_est_blanc,[Var_term("x")]),
                    F1_Exists("y",
                            F1_And(
                                F1_Atom(P2_plus_petit_que,[Var_term("x");Var_term("y")]),
                                F1_Atom(P2_a_l_est_de,[Var_term("y");Var_term("x")])))));;
let f5 = F1_Forall("x",
             F1_Or(F1_Atom(P1_a_l_est,[Var_term("x")]),
                  F1_Or(F1_Atom(P1_a_l_ouest,[Var_term("x")]),
                       F1_Or(F1_Atom(P1_au_sud,[Var_term("x")]),
                            F1_Atom(P1_au_nord,[Var_term("x")])))));;
let f6 = F1_And(
    F1_Forall("x",
            F1_Impl(F1_Atom(P1_est_grand,[Var_term("x")]),
                    F1_Atom(P1_est_rouge,[Var_term("x")]))),
    F1_Neg(F1_Exists("x",
                  F1_And(F1_Atom(P1_est_blanc,[Var_term("x")]),
                        F1_Exists("y",
                                F1_And(
                                    F1_Atom(P1_est_rouge,[Var_term("x")]),
                                    F1_Atom(P2_au_sud_de,[Var_term("x");Var_term("y")])))))));;
let f7 = F1_Exists("x",
             F1_And(F1_Atom(P1_est_rouge,[Var_term("x")]),
                    F1_Atom(P2_au_nord_de,[Var_term("x");Cons_term(F0_G,[])])));;
let f8 = F1_Exists("x",
             F1_And(
                 F1_And(F1_Atom(P1_Rose,[Var_term("x")]),
                       F1_Atom(P1_est_rouge,[Var_term("x")])),
                 F1_Forall("y",
                         F1_Impl(
                             F1_And(
                                 F1_Atom(P1_Rose,[Var_term("y")]),
                                 F1_Atom(P1_est_rouge,[Var_term("y")])),
                             F1_Atom(P2_egal,[Var_term("x");Var_term("y")])))));;
```

2.2 Sémantique : Interprétation des formules

2.2.1 Définition et implantation

Les formules de la logique des prédicats sont interprétées par une valeur booléenne ($\mathbb{B} = \{0, 1\}$). La notion de structure présentée plus haut est étendue afin de donner une interprétation aux symboles de $\mathcal{P}$. Une structure $\mathbf{M}$ est la donnée :

— d'un ensemble non vide $|\mathbf{M}|$, appelé le domaine d'interprétation de $\mathbf{M}$,
— d'une application qui associe à tout symbole $f \in \mathcal{F}$ d'arité n une application $f^{\mathbf{M}} : |\mathbf{M}|^n \to |\mathbf{M}|$,
— d'une application qui associe à tout symbole $p \in \mathcal{P}$ d'arité n une relation n-aire $p^{\mathbf{M}} \subseteq |\mathbf{M}|^n$.

Etant données une structure $\mathbf{M}$ et une valuation $\nu : X \to |\mathbf{M}|$, la valeur booléenne, notée $[\![\varphi]\!]_\nu^{\mathbf{M}}$, associée à une formule φ de la logique du premier ordre est définie comme suit.

$$
\begin{array}{l}
[\![true]\!]_\nu^{\mathbf{M}} = 1 \\
[\![false]\!]_\nu^{\mathbf{M}} = 0 \\
[\![p(t_1, \cdots, t_n)]\!]_\nu^{\mathbf{M}} = 1 \text{ si et seulement si } \left([\![t_1]\!]_\nu^{\mathbf{M}}, \cdots, [\![t_n]\!]_\nu^{\mathbf{M}}\right) \in p^{\mathbf{M}} \\
[\![\neg\psi]\!]_\nu^{\mathbf{M}} = \overline{[\![\psi]\!]_\nu^{\mathbf{M}}} \\
[\![\psi_1 \wedge \psi_2]\!]_\nu^{\mathbf{M}} = [\![\psi_1]\!]_\nu^{\mathbf{M}}.[\![\psi_2]\!]_\nu^{\mathbf{M}} \\
[\![\psi_1 \vee \psi_2]\!]_\nu^{\mathbf{M}} = [\![\psi_1]\!]_\nu^{\mathbf{M}} + [\![\psi_2]\!]_\nu^{\mathbf{M}} \\
[\![\psi_1 \Rightarrow \psi_2]\!]_\nu^{\mathbf{M}} = \overline{[\![\psi_1]\!]_\nu^{\mathbf{M}}} + [\![\psi_2]\!]_\nu^{\mathbf{M}} \\
[\![\forall x\ \varphi]\!]_\nu^{\mathbf{M}} = 1 \text{ si et seulement si } [\![\varphi]\!]_{\nu[x\leftarrow m]}^{\mathbf{M}} = 1 \text{ pour tout } m \in |\mathbf{M}| \\
[\![\exists x\ \varphi]\!]_\nu^{\mathbf{M}} = 1 \text{ si et seulement si il existe } m \in |\mathbf{M}| \text{ tel que } [\![\varphi]\!]_{\nu[x\leftarrow m]}^{\mathbf{M}} = 1
\end{array}
$$

où $\nu[x \leftarrow m]$ représente la valuation définie par :

$$\nu[x \leftarrow m] : X \to |\mathbf{M}| \qquad \nu[x \leftarrow m](v) = \begin{cases} m & \text{si } v = x \\ \nu(v) & \text{sinon} \end{cases}$$

Lorsque le domaine d'interprétation d'une structure $\mathbf{M}$ n'est pas fini, il n'est pas possible de calculer en temps fini $[\![\varphi]\!]_\nu^{\mathbf{M}}$ en suivant le schéma d'interprétation ci-dessus. En effet, interpréter des formules contenant au moins un quantificateur nécessite de parcourir l'ensemble des valeurs possibles du domaine et ne peut être effectué en temps fini lorsque l'ensemble $|\mathbf{M}|$ n'est pas fini. Aussi, la fonction `interp_f`, qui calcule $[\![\varphi]\!]_\nu^{\mathbf{M}}$, prend en argument les deux fonctions `check_forall` et `check_exists` qui, étant donnés un symbole de variable x et une formule ψ, permettent respectivement de déterminer si $[\![\psi]\!]_{\nu[x\leftarrow m]}^{\mathbf{M}} = 1$ pour toute valeur $m \in |\mathbf{M}|$, et de déterminer s'il existe au moins une valeur $m \in |\mathbf{M}|$ telle que $[\![\psi]\!]_{\nu[x\leftarrow m]}^{\mathbf{M}} = 1$. Nous donnons plus loin la définition de ces deux fonctions lorsque l'ensemble $|\mathbf{M}|$ est fini. Les autres paramètres de la fonction `interp_f` sont la relation d'égalité sur les symboles de variable (`eqX`), la valuation (`v`), l'application `i_f` qui associe la fonction $f^{\mathbf{M}}$ à tout symbole de fonction f, et l'application `i_p` qui associe la fonction caractéristique de la relation $p^{\mathbf{M}}$ à tout symbole de prédicat p. Comme dans la fonction `interp_term` d'interprétation des termes, aucun contrôle d'arité est effectué ici, et les n-uplets correspondant aux arguments de ces fonctions sont représentés par des listes.

Python

```python
def interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f):
    def _interp_t(t):
        return interp_term(v,i_f,t)
    if isinstance(f,F1_true):
        return True
    if isinstance(f,F1_false):
        return False
    if isinstance(f,F1_Atom):
        ip = i_p(f.symbpred)
        return ip([_interp_t(t) for t in f.lterms])
    if isinstance(f,F1_Neg):
        return not interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f)
    if isinstance(f,F1_And):
        return interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f1) \
               and interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f2)
    if isinstance(f,F1_Or):
        return interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f1) \
               or interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f2)
    if isinstance(f,F1_Impl):
        return (not interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f1)) \
               or interp_f(eqX,v,i_f,i_p,check_forall,check_exists,f.sub_f2)
```

```
    if isinstance(f,F1_Forall):
        return check_forall(eqX,v,i_f,i_p,f.symbvar,f.sub_f)
    if isinstance(f,F1_Exists):
        return check_exists(eqX,v,i_f,i_p,f.symbvar,f.sub_f)
    raise ValueError
```

Lorsque le domaine d'interprétation de $\mathbf{M}$ est un ensemble fini représenté par une liste `d`, il est possible de parcourir les éléments de cet ensemble et on peut alors définir les deux fonctions `check_forall` et `check_exists`, utilisées en paramètre de la fonction `interp_f`, comme suit (ces deux fonctions sont mutuellement récursives).

```
def finite_check_forall(d,eqX,v,i_f,i_p,x,f):
    def _forall(_eqX,_v,_i_f,_i_p,_x,_f):
        return finite_check_forall(d,_eqX,_v,_i_f,_i_p,_x,_f)
    def _exists(_eqX,_v,_i_f,_i_p,_x,_f):
        return finite_check_exists(d,_eqX,_v,_i_f,_i_p,_x,_f)
    for m in d:
        if not interp_f(eqX,change_val(eqX,v,x,m),i_f,i_p,_forall,_exists,f):
            return False
    return True

def finite_check_exists(d,eqX,v,i_f,i_p,x,f):
    def _forall(_eqX,_v,_i_f,_i_p,_x,_f):
        return finite_check_forall(d,_eqX,_v,_i_f,_i_p,_x,_f)
    def _exists(_eqX,_v,_i_f,_i_p,_x,_f):
        return finite_check_exists(d,_eqX,_v,_i_f,_i_p,_x,_f)
    for m in d:
        if interp_f(eqX,change_val(eqX,v,x,m),i_f,i_p,_forall,_exists,f):
            return True
    return False
```

où `change_val` est la fonction permettant de construire la valuation $\nu[x \leftarrow m]$ définie comme suit.

```
def change_val(eq,v,x,m):
    def _v(y):
        if eq(x,y):
            return m
        else:
            return v(y)
    return _v
```

OCaml

```
# let rec interp_f eqX v i_f i_p check_forall check_exists f =
 match f with
   F1_true -> true
 | F1_false -> false
 | F1_Atom(p,l) -> ((i_p p) (List.map (interp_term v i_f) l))
 | F1_Neg(f0) -> not (interp_f eqX v i_f i_p check_forall check_exists f0)
 | F1_And(f1,f2) -> (interp_f eqX v i_f i_p check_forall check_exists f1) &&
                    (interp_f eqX v i_f i_p check_forall check_exists f2)
 | F1_Or(f1,f2) -> (interp_f eqX v i_f i_p check_forall check_exists f1) ||
                   (interp_f eqX v i_f i_p check_forall check_exists f2)
 | F1_Impl(f1,f2) -> (not (interp_f eqX v i_f i_p check_forall check_exists f1))
                     || (interp_f eqX v i_f i_p check_forall check_exists f2)
 | F1_Forall(x,f0) -> (check_forall eqX v i_f i_p x f0)
 | F1_Exists(x,f0) -> (check_exists eqX v i_f i_p x f0);;
val interp_f : 'a -> ('b -> 'c) -> ('d -> 'c list -> 'c) -> ('e -> 'c list -> bool)
 -> ('a -> ('b -> 'c) -> ('d -> 'c list -> 'c) -> ('e -> 'c list -> bool) -> 'b
          -> ('b, 'd, 'e) formul1 -> bool)
 -> ('a -> ('b -> 'c) -> ('d -> 'c list -> 'c) -> ('e -> 'c list -> bool) -> 'b
```

```
          -> ('b, 'd, 'e) formul1 -> bool)
 -> ('b, 'd, 'e) formul1 -> bool = <fun>
```

Lorsque le domaine d'interprétation de $\mathbf{M}$ est un ensemble fini représenté par une liste `d`, il est possible de parcourir les éléments de cet ensemble et on peut alors définir les deux fonctions `check_forall` et `check_exists`, utilisées en paramètre de la fonction `interp_f`, comme suit (ces deux fonctions sont mutuellement récursives).

```
# let rec finite_check_forall d eqX v i_f i_p x f  =
 (List.for_all
  (function m -> (interp_f eqX (change_val eqX v x m) i_f i_p
                         (finite_check_forall d) (finite_check_exists d) f))
  d)
and finite_check_exists d eqX v i_f i_p x f  =
 (List.exists
  (function m -> (interp_f eqX (change_val eqX v x m) i_f i_p
                         (finite_check_forall d) (finite_check_exists d) f))
  d);;
val finite_check_forall : 'a list -> ('b -> 'b -> bool) -> ('b -> 'a) -> ('c -> 'a list -> 'a)
 -> ('d -> 'a list -> bool) -> 'b -> ('b, 'c, 'd) formul1 -> bool = <fun>
val finite_check_exists : 'a list -> ('b -> 'b -> bool) -> ('b -> 'a) -> ('c -> 'a list -> 'a)
 -> ('d -> 'a list -> bool) -> 'b -> ('b, 'c, 'd) formul1 -> bool = <fun>
```

où `change_val` est la fonction permettant de construire la valuation $\nu[x \leftarrow m]$ définie comme suit.

```
# let change_val eq v x m = function y -> if (eq x y) then m else (v y);;
val change_val : ('a -> 'b -> bool) -> ('b -> 'c) -> 'a -> 'c -> 'b -> 'c = <fun>
```

2.2.2 Interprétation des formules du projet

Construction d'une structure à partir d'un jardin L'interprétation des symboles de $\mathcal{F}_0$ utilisés dans les formules du projet est obtenue à partir de la donnée d'un jardin j et est définie dans la section 1.2.2. A partir d'un jardin j, l'interprétation des symboles de prédicat est définie comme suit.
Prédicats d'espèce Les trois prédicats Rose, Paquerette et Tulipe sont interprétés par des relations unaires caractérisant les places où sont situées les différentes espèces de fleur.

$$\begin{array}{rcl} \mathrm{Rose}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), \mathrm{rose}, t, c, n) \in j\} \\ \mathrm{Paquerette}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), \mathrm{paquerette}, t, c, n) \in j\} \\ \mathrm{Tulipe}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), \mathrm{tulipe}, t, c, n) \in j\} \end{array}$$

Prédicats de taille Les trois prédicats est_grand, est_moyen et est_petit sont interprétés par des relations unaires caractérisant les places où sont situées les différentes tailles de fleur.

$$\begin{array}{rcl} \mathrm{est\_grand}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), e, \mathrm{grand}, c, n) \in j\} \\ \mathrm{est\_moyen}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), e, \mathrm{moyen}, c, n) \in j\} \\ \mathrm{est\_petit}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), e, \mathrm{petit}, c, n) \in j\} \end{array}$$

Prédicats de couleur Les trois prédicats est_rouge, est_rose et est_blanc sont interprétés par des relations unaires caractérisant les places où sont situées les différentes couleurs de fleur.

$$\begin{array}{rcl} \mathrm{est\_rouge}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), e, t, \mathrm{rouge}, n) \in j\} \\ \mathrm{est\_rose}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), e, t, \mathrm{rose}, n) \in j\} \\ \mathrm{est\_blanc}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid ((x,y), e, t, \mathrm{blanc}, n) \in j\} \end{array}$$

Prédicats unaires de position Les quatre prédicats a_l_est, a_l_ouest, au_sud et au_nord sont interprétés par des relations unaires caractérisant les places appartenant aux zones géographiques délimitées sur la figure 1.

$$\begin{array}{rcl}
\text{a_l_est}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid x > 0 \text{ et } |x| > |y|\} \\
\text{a_l_ouest}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid x < 0 \text{ et } |x| > |y|\} \\
\text{au_sud}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid y < 0 \text{ et } |y| > |x|\} \\
\text{au_nord}^{\mathbf{M}_j} & = & \{(x,y) \in |\mathbf{M}_j| \mid y > 0 \text{ et } |y| > |x|\}
\end{array}$$

Prédicats binaires de comparaison de positions Les six prédicats a_l_est_de, a_l_ouest_de, au_sud_de, au_nord_de, meme_latitude et meme_longitude sont interprétés par des relations binaires caractérisant des couples de places.

$$\begin{array}{rcl}
\text{a_l_est_de}^{\mathbf{M}_j} & = & \{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid x_1 > x_2\} \\
\text{a_l_ouest_de}^{\mathbf{M}_j} & = & \{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid x_1 < x_2\} \\
\text{au_sud_de}^{\mathbf{M}_j} & = & \{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid y_1 < y_2\} \\
\text{au_nord_de}^{\mathbf{M}_j} & = & \{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid y_1 > y_2\} \\
\text{meme_latitude}^{\mathbf{M}_j} & = & \{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid y_1 = y_2\} \\
\text{meme_longitude}^{\mathbf{M}_j} & = & \{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid x_1 = x_2\}
\end{array}$$

Prédicats de comparaison de tailles Les trois prédicats plus_grand_que, plus_petit_que et meme_taille_que sont interprétés par des relations binaires caractérisant des couples de places.

$$\begin{array}{rcl}
\text{plus_grand_que}^{\mathbf{M}_j} & = & \left\{ \begin{array}{ll}
\multicolumn{2}{l}{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid} \\
 & ((x_1,y_1),e_1,t_1,c_1,n_1) \in j \\
\text{et} & ((x_2,y_2),e_2,t_2,c_2,n_2) \in j \\
\text{et} & \left(\begin{array}{ll} & (t_1 = \text{grand et } (t_2 = \text{moyen ou } t_2 = \text{petit})) \\ \text{ou} & (t_1 = \text{moyen et } t_2 = \text{petit}) \end{array} \right)
\end{array} \right\} \\
\text{plus_petit_que}^{\mathbf{M}_j} & = & \left\{ \begin{array}{ll}
\multicolumn{2}{l}{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid} \\
 & ((x_1,y_1),e_1,t_1,c_1,n_1) \in j \\
\text{et} & ((x_2,y_2),e_2,t_2,c_2,n_2) \in j \\
\text{et} & \left(\begin{array}{ll} & (t_1 = \text{petit et } (t_2 = \text{moyen ou } t_2 = \text{grand})) \\ \text{ou} & (t_1 = \text{moyen et } t_2 = \text{grand}) \end{array} \right)
\end{array} \right\} \\
\text{meme_taille_que}^{\mathbf{M}_j} & = & \left\{ \begin{array}{ll}
\multicolumn{2}{l}{((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid} \\
 & ((x_1,y_1),e_1,t_1,c_1,n_1) \in j \\
\text{et} & ((x_2,y_2),e_2,t_2,c_2,n_2) \in j \\
\text{et} & t_1 = t_2
\end{array} \right\}
\end{array}$$

Prédicat de comparaison de couleurs Le prédicat meme_couleur_que est interprété par une relation binaire caractérisant des couples de places.

$$\text{meme_couleur_que}^{\mathbf{M}_j} = \left\{ \begin{array}{l}
((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid \\
\quad ((x_1,y_1),e,t_1,c_1,n_1) \in j \text{ et } ((x_2,y_2),e_2,t_2,c_2,n_2) \in j \\
\quad \text{et } c_1 = c_2
\end{array} \right\}$$

Prédicat d'égalité Le prédicat $\mathrel{\hat{=}}$ est interprété par une relation binaire caractérisant des couples de places.

$$\mathrel{\hat{=}}^{\mathbf{M}_j} = \left\{ \; ((x_1,y_1),(x_2,y_2)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \mid x_1 = x_2 \text{ et } y_1 = y_2 \; \right\}$$

Prédicat ternaire de comparaison de positions Le prédicat est_entre est interprété par une relation ternaire caractérisant des triplets de places.

$$\text{est_entre}^{\mathbf{M}_j} = \left\{ \begin{array}{l}
((x_1,y_1),(x_2,y_2),(x_3,y_3)) \in |\mathbf{M}_j| \times |\mathbf{M}_j| \times |\mathbf{M}_j| \mid \\
0 \leq \dfrac{x_1 - x_2}{x_3 - x_2} \text{ et } \dfrac{x_1 - x_2}{x_3 - x_2} = \dfrac{y_1 - y_2}{y_3 - y_2} \text{ et } \dfrac{y_1 - y_2}{y_3 - y_2} \leq 1
\end{array} \right\}$$

Le point de coordonnées (x_1, y_1) se trouve sur la même droite que les points de coordonnées (x_2, y_2) et (x_3, y_3) et se situe entre ces deux points.

Python

L'interprétation des symboles de $\mathcal{P}$ s'obtient comme suit :

```
def j_interp_p(j):
    def _chercher_dans_jardin(z):
        for (p,e,t,c,n) in j:
            if p==z:
                return (e,t,c,n)
        raise ValueError
    def _interp_p(p):
        def __interp_Rose(l):
            if len(l)==1:
                (e,_,_,_) = _chercher_dans_jardin(l[0])
                return e=="rose"
            raise ValueError
        def __interp_Paquerette(l):
            if len(l)==1:
                (e,_,_,_) = _chercher_dans_jardin(l[0])
                return e=="paquerette"
            raise ValueError
        def __interp_Tulipe(l):
            if len(l)==1:
                (e,_,_,_) = _chercher_dans_jardin(l[0])
                return e=="tulipe"
            raise ValueError
        def __interp_est_grand(l):
            if len(l)==1:
                (_,t,_,_) = _chercher_dans_jardin(l[0])
                return t=="grand"
            raise ValueError
        def __interp_est_moyen(l):
            if len(l)==1:
                (_,t,_,_) = _chercher_dans_jardin(l[0])
                return t=="moyen"
            raise ValueError
        def __interp_est_petit(l):
            if len(l)==1:
                (_,t,_,_) = _chercher_dans_jardin(l[0])
                return t=="petit"
            raise ValueError
        def __interp_est_rouge(l):
            if len(l)==1:
                (_,_,c,_) = _chercher_dans_jardin(l[0])
                return c=="rouge"
            raise ValueError
        def __interp_est_rose(l):
            if len(l)==1:
                (_,_,c,_) = _chercher_dans_jardin(l[0])
                return c=="rose"
            raise ValueError
        def __interp_est_blanc(l):
            if len(l)==1:
                (_,_,c,_) = _chercher_dans_jardin(l[0])
                return c=="blanc"
            raise ValueError
        def __interp_a_l_est(l):
            if len(l)==1:
                return (l[0][0] > 0) and (abs(l[0][0]) > abs(l[0][1]))
            raise ValueError
        def __interp_a_l_ouest(l):
            if len(l)==1:
                return (l[0][0] < 0) and (abs(l[0][0]) > abs(l[0][1]))
            raise ValueError
```

```
    def __interp_au_sud(l):
        if len(l)==1:
            return (l[0][1] < 0) and (abs(l[0][1]) > abs(l[0][0]))
        raise ValueError
    def __interp_au_nord(l):
        if len(l)==1:
            return (l[0][1] > 0) and (abs(l[0][1]) > abs(l[0][0]))
        raise ValueError
    def __interp_a_l_est_de(l):
        if len(l)==2:
            return l[0][0]>l[1][0]
        raise ValueError
    def __interp_a_l_ouest_de(l):
        if len(l)==2:
            return l[0][0]<l[1][0]
        raise ValueError
    def __interp_au_sud_de(l):
        if len(l)==2:
            return l[0][1]<l[1][1]
        raise ValueError
    def __interp_au_nord_de(l):
        if len(l)==2:
            return l[0][1]>l[1][1]
        raise ValueError
    def __interp_meme_latitude(l):
        if len(l)==2:
            return l[0][1]==l[1][1]
        raise ValueError
    def __interp_meme_longitude(l):
        if len(l)==2:
            return l[0][0]==l[1][0]
        raise ValueError
    def __interp_plus_grand_que(l):
        if len(l)==2:
            (_,t1,_,_) = _chercher_dans_jardin(l[0])
            (_,t2,_,_) = _chercher_dans_jardin(l[1])
            return (t1=="grand" and (t2=="moyen" or t2=="petit")) or \
                   (t1=="moyen" and t2=="petit")
        raise ValueError
    def __interp_plus_petit_que(l):
        if len(l)==2:
            (_,t1,_,_) = _chercher_dans_jardin(l[0])
            (_,t2,_,_) = _chercher_dans_jardin(l[1])
            return (t1=="petit" and (t2=="moyen" or t2=="grand")) or \
                   (t1=="moyen" and t2=="grand")
        raise ValueError
    def __interp_meme_taille_que(l):
        if len(l)==2:
            (_,t1,_,_) = _chercher_dans_jardin(l[0])
            (_,t2,_,_) = _chercher_dans_jardin(l[1])
            return t1==t2
        raise ValueError
    def __interp_meme_couleur_que(l):
        if len(l)==2:
            (_,_,c1,_) = _chercher_dans_jardin(l[0])
            (_,_,c2,_) = _chercher_dans_jardin(l[1])
            return c1==c2
        raise ValueError
    def __interp_egal(l):
        if len(l)==2:
            return l[0]==l[1]
        raise ValueError
    def __interp_est_entre(l):
```

```
        if len(l)==3:
            (x1,y1)=l[0]
            (x2,y2)=l[1]
            (x3,y3)=l[2]
            f1=(x1-x2)/(x3-x2)
            f2=(y1-y2)/(y3-y2)
            return (0 <= f1) and (f1==f2) and (f2 <= 1)
        raise ValueError
    if p=="Rose":
        return __interp_Rose
    if p=="Paquerette":
        return __interp_Paquerette
    if p=="Tulipe":
        return __interp_Tulipe
    if p=="est_grand":
        return __interp_est_grand
    if p=="est_moyen":
        return __interp_est_moyen
    if p=="est_petit":
        return __interp_est_petit
    if p=="est_rouge":
        return __interp_est_rouge
    if p=="est_rose":
        return __interp_est_rose
    if p=="est_blanc":
        return __interp_est_blanc
    if p=="a_l_est":
        return __interp_a_l_est
    if p=="a_l_ouest":
        return __interp_a_l_ouest
    if p=="au_sud":
        return __interp_au_sud
    if p=="au_nord":
        return __interp_au_nord
    if p=="a_l_est_de":
        return __interp_a_l_est_de
    if p=="a_l_ouest_de":
        return __interp_a_l_ouest_de
    if p=="au_sud_de":
        return __interp_au_sud_de
    if p=="au_nord_de":
        return __interp_au_nord_de
    if p=="meme_latitude":
        return __interp_meme_latitude
    if p=="meme_longitude":
        return __interp_meme_longitude
    if p=="plus_grand_que":
        return __interp_plus_grand_que
    if p=="plus_petit_que":
        return __interp_plus_petit_que
    if p=="meme_taille_que":
        return __interp_meme_taille_que
    if p=="meme_couleur_que":
        return __interp_meme_couleur_que
    if p=="egal":
        return __interp_egal
    if p=="est_entre":
        return __interp_est_entre
    raise ValueError
return _interp_p
```

Etant donné un jardin `j`, `j_interp_p(j)` retourne la fonction `_interp_p` qui correspond à une fonction qui étant donné un symbole de prédicat `p` retourne la fonction qui correspond à l'interprétation de ce symbole de prédicat. Cette interprétation correspond à une fonction qui étant donnée une liste `l` déclenche une erreur si cette liste ne contient pas le nombre d'arguments correspondant à l'arité du prédicat `p`, et sinon retourne le booléen calculé à partir des éléments de `l`. Par exemple, si l'on considère le jardin défini page 5 et le prédicat `plus_petit_que`, on a :

```
mon_interp_p = j_interp_p(mon_jardin)
mon_interp_plus_petit_que = mon_interp_p("plus_petit_que")

>>> mon_interp_plus_petit_que([(-3,2),(-2,-1)])
True
>>> mon_interp_plus_petit_que([(4,-3),(6,-1)])
False
```

OCaml

L'interprétation des symboles de $\mathcal{P}$ s'obtient comme suit :

```
# let j_interp_p j p =
 let rec chercher_dans_jardin z cj = match cj with
    [] -> failwith("ValueError")
  | (p,e,t,c,n)::r -> if p=z then (e,t,c,n) else (chercher_dans_jardin z r)
in match p with
P1_Rose
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                            (E_rose,_,_,_) -> true
                          | _ -> false)
            | _ -> failwith("ValueError"))
| P1_Paquerette
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                            (E_paquerette,_,_,_) -> true
                          | _ -> false)
            | _ -> failwith("ValueError"))
| P1_Tulipe
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                            (E_tulipe,_,_,_) -> true
                          | _ -> false)
            | _ -> failwith("ValueError"))
| P1_est_grand
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                            (_,T_grand,_,_) -> true
                          | _ -> false)
            | _ -> failwith("ValueError"))
| P1_est_moyen
-> (function l -> match l with
              [a] -> (match (chercher_dans_jardin a j) with
                             (_,T_moyen,_,_) -> true
                           | _ -> false)
             | _ -> failwith("ValueError"))
| P1_est_petit
-> (function l -> match l with
              [a] -> (match (chercher_dans_jardin a j) with
                             (_,T_petit,_,_) -> true
                           | _ -> false)
             | _ -> failwith("ValueError"))
| P1_est_rouge
```

```
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                           (_,_,C_rouge,_) -> true
                         | _ -> false)
            | _ -> failwith("ValueError"))
| P1_est_rose
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                           (_,_,C_rose,_) -> true
                         | _ -> false)
            | _ -> failwith("ValueError"))
| P1_est_blanc
-> (function l -> match l with
             [a] -> (match (chercher_dans_jardin a j) with
                           (_,_,C_blanc,_) -> true
                         | _ -> false)
            | _ -> failwith("ValueError"))
| P1_a_l_est
-> (function l -> match l with
     [(x,y)] -> (x > 0) && ((abs x) > (abs y))
           | _ -> failwith("ValueError"))
| P1_a_l_ouest
-> (function l -> match l with
             [(x,y)] -> (x < 0) && ((abs x) > (abs y))
           | _ -> failwith("ValueError"))
| P1_au_sud
-> (function l -> match l with
             [(x,y)] -> (y < 0) && ((abs y) > (abs x))
           | _ -> failwith("ValueError"))
| P1_au_nord
-> (function l -> match l with
             [(x,y)] -> (y > 0) && ((abs y) > (abs x))
           | _ -> failwith("ValueError"))
| P2_a_l_est_de
-> (function l -> match l with
             [(x1,y1);(x2,y2)] -> (x1 > x2)
           | _ -> failwith("ValueError"))
| P2_a_l_ouest_de
-> (function l -> match l with
             [(x1,y1);(x2,y2)] -> (x1 < x2)
           | _ -> failwith("ValueError"))
| P2_au_sud_de
-> (function l -> match l with
             [(x1,y1);(x2,y2)] -> (y1 < x2)
           | _ -> failwith("ValueError"))
| P2_au_nord_de
-> (function l -> match l with
             [(x1,y1);(x2,y2)] -> (y1 > y2)
           | _ -> failwith("ValueError"))
| P2_meme_latitude
-> (function l -> match l with
             [(x1,y1);(x2,y2)] -> (y1 = y2)
           | _ -> failwith("ValueError"))
| P2_meme_longitude
-> (function l -> match l with
             [(x1,y1);(x2,y2)] -> (x1 = x2)
           | _ -> failwith("ValueError"))
| P2_plus_grand_que
-> (function l -> match l with
             [a1;a2] -> (match ((chercher_dans_jardin a1 j),(chercher_dans_jardin a2 j)) with
                             ((_,T_grand,_,_),(_,T_moyen,_,_)) -> true
                           | ((_,T_grand,_,_),(_,T_petit,_,_)) -> true
                           | ((_,T_moyen,_,_),(_,T_petit,_,_)) -> true
```

```
         | _ -> false)
            | _ -> failwith("ValueError"))
| P2_plus_petit_que
-> (function l -> match l with
             [a1;a2] -> (match ((chercher_dans_jardin a1 j),(chercher_dans_jardin a2 j)) with
                               ((_,T_petit,_,_),(_,T_moyen,_,_)) -> true
                             | ((_,T_petit,_,_),(_,T_grand,_,_)) -> true
                             | ((_,T_moyen,_,_),(_,T_grand,_,_)) -> true
     | _ -> false)
            | _ -> failwith("ValueError"))
| P2_meme_taille_que
-> (function l -> match l with
             [a1;a2] -> (match ((chercher_dans_jardin a1 j),(chercher_dans_jardin a2 j)) with
                               ((_,t1,_,_),(_,t2,_,_)) -> t1=t2)
             | _ -> failwith("ValueError"))
| P2_meme_couleur_que
-> (function l -> match l with
             [a1;a2] -> (match ((chercher_dans_jardin a1 j),(chercher_dans_jardin a2 j)) with
                               ((_,_,c1,_),(_,_,c2,_)) -> c1=c2)
           | _ -> failwith("ValueError"))
| P2_egal
-> (function l -> match l with
             [a1;a2] -> a1=a2
           | _ -> failwith("ValueError"))
| P3_est_entre
-> (function l -> match l with
     [(x1,y1);(x2,y2);(x3,y3)]
     -> (let f1 = (float_of_int (x1 - x2)) /. (float_of_int (x3 - x2)) in
 let f2 = (float_of_int (y1 - y2)) /. (float_of_int (y3 - y2)) in
 (0. <= f1) && (f1=f2) && (f2 <= 1.))
             | _ -> failwith("ValueError"));;
val j_interp_p : ((int * int) * especes * tailles * couleurs * 'a) list
                 -> sig_p -> (int * int) list -> bool = <fun>
```

Etant donné un jardin `j`, `(j_interp_p j)` correspond à une fonction qui étant donné un symbole de prédicat `p` retourne la fonction qui correspond à l'interprétation de ce symbole de prédicat. Cette interprétation correspond à une fonction qui étant donnée une liste `l` déclenche une erreur si cette liste ne contient pas le nombre d'arguments correspondant à l'arité du prédicat `p`, et sinon retourne le booléen calculé à partir des éléments de `l`. Par exemple, si l'on considère le jardin défini page 5 et le prédicat `P2_plus_petit_que`, on a :

```
# let mon_interp_p = (j_interp_p mon_jardin);;
val mon_interp_p : sig_p -> (int * int) list -> bool = <fun>
# let mon_interp_plus_petit_que = (mon_interp_p P2_plus_petit_que);;
val mon_interp_plus_petit_que : (int * int) list -> bool = <fun>
# (mon_interp_plus_petit_que [(-3,2);(-2,-1)]);;
- : bool = true
# (mon_interp_plus_petit_que [(4,-3);(6,-1)]);;
- : bool = false
```

Interprétation des formules Dans le cadre du projet, toutes les formules soumises à la vérification sont des formules closes (une erreur est donc déclenchée lors de l'évaluation d'une formule contenant au moins une variable libre).

Python

L'évaluation d'une formule s'obtient simplement à partir des définitions introduites ci-dessus.

```
def j_interp_formul(j):
    d = j_domain(j)
    i_f = j_interp_f(j)
```

```
    i_p = j_interp_p(j)
    def _j_forall():
        def _forall(_eqX,_v,_i_f,_i_p,_x,_f):
            return finite_check_forall(d,_eqX,_v,_i_f,_i_p,_x,_f)
        return _forall
    def _j_exists():
        def _exists(_eqX,_v,_i_f,_i_p,_x,_f):
            return finite_check_exists(d,_eqX,_v,_i_f,_i_p,_x,_f)
        return _exists
    def _interp_formul(f):
        if ground(eq_atom,f):
            return interp_f(eq_atom,valuation_error,i_f,i_p,_j_forall(),_j_exists(),f)
        else:
            raise ValueError
    return _interp_formul
```

Par exemple, si l'on considère à nouveau le jardin défini page 5, on peut construire la fonction d'évaluation des formules comme suit, et l'utiliser pour évaluer les huit formules introduites ci-dessus.

```
mon_interp_formul = j_interp_formul(mon_jardin)
>>> mon_interp_formul(f1)
False
>>> mon_interp_formul(f2)
False
>>> mon_interp_formul(f3)
True
>>> mon_interp_formul(f4)
True
>>> mon_interp_formul(f5)
True
>>> mon_interp_formul(f6)
True
>>> mon_interp_formul(f7)
False
>>> mon_interp_formul(f8)
True
```

OCaml

L'évaluation d'une formule s'obtient simplement à partir des définitions introduites ci-dessus.

```
# let j_interp_formul j =
 let d = (j_domain j) in
 let i_f = (j_interp_f j) in
 let i_p = (j_interp_p j) in
 function f -> if (ground eq_atom f)
              then (interp_f eq_atom valuation_error i_f i_p
                             (finite_check_forall d)
                             (finite_check_exists d) f)
              else failwith("ValueError");;
val j_interp_formul : ((int * int) * especes * tailles * couleurs * 'a option) list
    -> ('b, 'a, sig_p) formul1 -> bool = <fun>
```

Par exemple, si l'on considère à nouveau le jardin défini page 5, on peut construire la fonction d'évaluation des formules comme suit, et l'utiliser pour évaluer les huit formules introduites ci-dessus.

```
# let mon_interp_formul = (j_interp_formul mon_jardin);;
val mon_interp_formul : ('_a, sig_f0, sig_p) formul1 -> bool = <fun>
# (mon_interp_formul f1);;
# - : bool = false
```

```
# (mon_interp_formul f2);;
# - : bool = false
# (mon_interp_formul f3);;
# - : bool = true
# (mon_interp_formul f4);;
# - : bool = true
# (mon_interp_formul f5);;
# - : bool = true
# (mon_interp_formul f6);;
# - : bool = true
# (mon_interp_formul f7);;
# - : bool = false
# (mon_interp_formul f8);;
# - : bool = true
```

A Opérations ensemblistes

Représentation d'un ensemble fini Lorsque E est un ensemble fini contenant n éléments, il est possible de le représenter en extension en utilisant une structure de données permettant de réunir un nombre fini d'éléments. On peut par exemple utiliser une liste $[e_1, \cdots, e_n]$ contenant les n éléments de E dans un ordre correspondant à une certaine énumération de ses éléments. On choisit ici de représenter un ensemble fini par une liste dans laquelle un élément ne peut pas apparaître plus d'une fois (toutes les listes ne représentent donc pas un ensemble, par exemple la liste $[1, 1, 1]$ ne correspond pas à la représentation d'un ensemble par une liste sans ≪ doublon ≫). Il existe bien sûr plusieurs énumérations possibles pour un ensemble fini, chacune correspondant à une permutation (sans répétition) de ses éléments. Il existe $n! = 1 \times 2 \times \cdots \times n$ permutations différentes des éléments appartenant à un ensemble fini E de cardinal n. Toutes ces permutations représentent le même ensemble puisque l'ordre d'apparition des éléments dans la définition en extension d'un ensemble n'est pas discriminant.

Toutes les fonctions sur les ensembles que nous définissons ici sont paramétrées par la relation d'égalité sur les éléments de cet ensemble. Si l'on choisit une représentation par une liste ≪ sans doublon ≫ d'un ensemble fini, l'ensemble vide correspond à une liste vide, le singleton $\{e\}$ est obtenu en construisant la liste $[e]$ et la relation d'appartenance à un ensemble se ramène à la relation d'appartenance à une liste.

Python

```
emptyset = []
def singleton(e):
    return [e]
def is_in(eq,x,e):
    for h in e:
        if eq(x,h):
            return True
    return False
```

OCaml

```
# let emptyset = [];;
val emptyset : 'a list = []
# let singleton e = [e];;
val singleton : 'a -> 'a list = <fun>
# let rec is_in eq x e = match e with [] -> false | h::t -> (eq x h) || (is_in eq x t);;
val is_in : ('a -> 'b -> bool) -> 'a -> 'b list -> bool = <fun>
```

Union de deux ensembles L'union de deux ensembles A et B, notée $A \cup B$, est l'ensemble des éléments appartenant à A ou appartenant à B :

$$A \cup B = \{e \mid e \in A \text{ ou } e \in B\}$$

Lorsque les ensembles A et B sont finis et sont représentés par des listes, il est possible de construire une liste représentant l'ensemble $A \cup B$ en ajoutant à la liste représentant B tous les éléments de la liste représentant A. Il faut bien sûr s'assurer que les éléments ajoutés ne figuraient pas déjà dans la liste afin d'éviter qu'un élément apparaisse plusieurs fois dans une liste représentant un ensemble. On définit donc tout d'abord une fonction `union_singleton` permettant d'effectuer cet ajout.

Python

```
def union_singleton(eq,x,le):
    if is_in(eq,x,le):
        return le
    else:
        return [x]+le
>>> union_singleton(eq_atom,2,[1,2,3])
[1, 2, 3]
>>> union_singleton(eq_atom,5,[1,2,3])
[5, 1, 2, 3]
def union(eq,la,lb):
    r = lb
    for a in la:
        r = union_singleton(eq,a,r)
    return r
>>> union(eq_atom,[1,2,3],[2,3,4])
[1, 2, 3, 4]
```

OCaml

```
# let union_singleton eq x le = if (is_in eq x le) then le else x::le;;
val union_singleton : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
# (union_singleton eq_atom 2 [1;2;3]);;
- : int list = [1; 2; 3]
# (union_singleton eq_atom 5 [1;2;3]);;
- : int list = [5; 1; 2; 3]
# let union eq lA lB = (List.fold_left (function x -> function y -> (union_singleton eq y x)) lB lA);;
val union : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list = <fun>
# (union eq_atom [1;2;3] [2;3;4]);;
- : int list = [1; 2; 3; 4]
```

Il est aussi possible d'étendre cette définition afin de calculer l'union d'un nombre fini quelconque d'ensembles : la fonction `union_sets` réalise cette opération à partir de la liste `ll` passée en argument et contenant les ensembles sur lesquels on souhaite appliquer l'union.

Python

```
def union_sets(eq,ll):
    r = []
    for l in ll:
        r = union(eq,l,r)
    return r
>>> union_sets(eq_atom,[[1],[1,2],[2,3,4],[3,4,5]])
[5, 4, 3, 2, 1]
```

OCaml

```
# let rec union_sets eq ll = match ll with [] -> [] | l::t -> (union eq l (union_sets eq t));;
val union_sets : ('a -> 'a -> bool) -> 'a list list -> 'a list = <fun>
# (union_sets eq_atom [[1];[1;2];[2;3;4];[3;4;5]]);;
- : int list = [1; 2; 5; 4; 3]
```

Différence La différence $A \backslash B$ est l'ensemble des éléments appartenant à A qui n'appartiennent pas à B :

$$A \backslash B = \{e \mid e \in A \text{ et } e \notin B\}$$

Lorsque les ensembles A et B sont finis et sont représentés par des listes, il est possible de construire une liste représentant l'ensemble $A \backslash B$ en conservant dans la liste représentant A uniquement les éléments n'appartenant pas à la liste représentant B.

Python

```
def diff_set(eq,lA,lB):
    return [a for a in lA if not is_in(eq,a,lB)]
>>> diff_set(eq_atom,[0,1,2,3,8],[2,3,4])
[0, 1, 8]
```

OCaml

```
# let diff_set eq lA lB = (List.filter (function x -> (not (is_in eq x lB))) lA);;
val diff_set : ('a -> 'b -> bool) -> 'a list -> 'b list -> 'a list = <fun>
# (diff_set eq_atom [0;1;2;3;8] [2;3;4]);;
- : int list = [0; 1; 8]
```