Exercices de programmation autour de la logique

2003/2004

Mathieu Jaume
Mathieu.Jaume@lip6.fr

Contents

1 Logique des propositions

1.1 Formules de la logique des propositions

L'ensemble $\mathcal{F}_p$ des formules de la logique des propositions est défini à partir d'un ensemble Σ_p de symboles de propositions et d'un ensemble Σ_c de connecteurs logiques permettant de construire des formules à partir d'autres formules. Un formule de la logique des propositions est donc un élément de $(\Sigma_p \cup \Sigma_c)^\star$, c'est à dire une suite finie de symboles appartenant à $\Sigma_p \cup \Sigma_c$. Toutefois, tous les éléments de cet ensemble ne sont pas des formules de la logique des propositions. Par exemple, si p et q sont des symboles de proposition et si l'ensemble Σ_c contient le connecteur binaire $\wedge$, pq n'est pas une formule alors que $p \wedge q$ est une formule (si les formules sont écrites en forme infixe). C'est l'analyse syntaxique qui permet de déterminer si un élément de $(\Sigma_p \cup \Sigma_c)^\star$ est une formule. Plusieurs techniques coexistent pour résoudre ce type de problèmes et relèvent d'un cours de compilation. Dans tout ce qui suit, on considèrera qu'une telle analyse a déja été effectuée. La *syntaxe abstraite* des formules de $\mathcal{F}_p$ est souvent donnée sous la forme d'une "règle de grammaire" BNF (*Bachus Naur Form*) comme suit :

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi$$

Cette règle exprime qu'une formule est soit un symbole de proposition, soit une formule obtenue en appliquant un connecteur logique sur d'autres formules. Dans ce qui suit on notera p, q, r, ... les symboles propositionnels et on considèrera l'ensemble $\Sigma_c = \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ des opérateurs logiques usuels. Excepté l'opérateur $\neg$ de négation qui est un connecteur unaire, tous ces connecteurs sont binaires (i.e., ils s'appliquent sur 2 formules). En fait, l'ensemble $\mathcal{F}_c$ est défini de manière inductive à partir de $\Sigma_p \cup \Sigma_c$ comme suit :

- tout élément de Σ_p est dans $\mathcal{F}_p$
- si $\varphi \in \mathcal{F}_p$ alors $\neg\varphi \in \mathcal{F}_p$
- si $\varphi, \psi \in \mathcal{F}_p$ alors $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$ et $\varphi \Leftrightarrow \psi$ sont des éléments de $\mathcal{F}_p$.

Par exemple, $p \Leftrightarrow (q \vee r)$ est une formule de $\mathcal{F}_p$. On remarquera que des parenthèses apparaissent dans cette formule alors qu'elles ne sont pas présentes dans l'ensemble des symboles utilisés pour définir $\mathcal{F}_p$: en fait, la définition de $\mathcal{F}_p$ exprime la syntaxe abstraite des formules qui sont donc vues au travers de leurs arbres de syntaxe abstraite. Les symboles de parenthèses font partie de la syntaxe concrète (utilisée avant l'analyse syntaxique) et permettent d'indiquer sans ambiguité sur quels arguments portent les connecteurs. Ici, nous considèrerons que l'analyse syntaxique des formules a déjà été effectuée et nous utiliserons les parenthèses uniquement pour indiquer sur quels arguments s'appliquent les connecteurs sans avoir à représenter l'arbre de syntaxe abstraite.

S'agissant d'une définition inductive, les formules de $\mathcal{F}_p$ sont exactement les théorèmes du système d'inférence du tableau 1. Les jugements sur lesquels portent les règles de ce système sont des suites quelconques de symboles appartenant à l'ensemble $\Sigma = \Sigma_p \cup \Sigma_c$. Par exemple, $q\neg\vee p$ est un jugement qui n'admet pas d'arbre de dérivation (ce qui correspond au fait qu'il ne s'agit pas d'une formule), alors que le jugement $p \Leftrightarrow (q \vee r)$ admet l'arbre de dérivation suivant :

$$(F_6) : \frac{(F_1) : \frac{}{p} \quad (F_4) : \frac{(F_1) : \frac{}{q} \quad (F_1) : \frac{}{r}}{q \vee r}}{p \Leftrightarrow (q \vee r)}$$

$(F_1) : \dfrac{}{p}\ (p \in \Sigma_p)$	$(F_2) : \dfrac{\varphi}{\neg\varphi}$	$(F_3) : \dfrac{\varphi \quad \psi}{\varphi \wedge \psi}$
$(F_4) : \dfrac{\varphi \quad \psi}{\varphi \vee \psi}$	$(F_5) : \dfrac{\varphi \quad \psi}{\varphi \Rightarrow \psi}$	$(F_6) : \dfrac{\varphi \quad \psi}{\varphi \Leftrightarrow \psi}$

Table 1: Syntaxe abstraite des formules de $\mathcal{F}_p$

Il s'agit en fait de l'arbre de syntaxe abstraite de la formule $p \Leftrightarrow (q \vee r) \in \mathcal{F}_p$.

Pour représenter les formules de $\mathcal{F}_p$ en OCAML, on définit le type suivant :

```
type 'a formul =
   Prop  of 'a                  | Neg   of 'a formul
  | Impl  of 'a formul*'a formul | And   of 'a formul*'a formul
  | Or    of 'a formul*'a formul | Equiv of 'a formul*'a formul ;;
```

Le constructeur `Prop` permet d'obtenir une formule à partir d'un symbole propositionnel de Σ_p : le type de ces symboles n'est donc pas fixé (il s'agit de la variable de type `'a` et le type des formules est donc polymorphe). Les autres constructeurs correspondent aux éléments de Σ_c.

Par exemple, si on choisit d'indexer les symboles propositionnels par des entiers, pour définir la formule $\neg(p_1 \Leftrightarrow (p_2 \Rightarrow p_3))$, on écrira :

```
# let f = Neg(Equiv(Prop(1),(Impl(Prop(2),Prop(3)))));;
```

`f` sera alors de type `int formul`.

1.2 Formes normales, Clauses

Comme nous le verrons, il existe essentiellement deux catégories de méthodes permettant de déterminer si une formule est "vraie" : soit on en établit une preuve, soit on se base sur l'interprétation de cette formule. Dans les deux cas nous aurons parfois besoin de contraindre la forme des formules considérées afin de pouvoir appliquer certaines méthodes. Toute formule $\varphi \in \mathcal{F}_p$ peut être transformée en une formule d'une certaine forme, dite forme normale, qui lui est "logiquement" équivalente. Nous verrons par la suite ce que peut signifier l'équivalence "logique". Plusieurs formes normales peuvent être envisagées : on considère ici les formes normales disjonctives et conjonctives. Tout d'abord, nous dirons qu'une formule est un ***littéral*** si elle s'écrit p ou $\neg p$ où p est un symbole propositionnel. Dans le premier cas, nous dirons qu'il s'agit d'un ***littéral positif***, tandis que dans le deuxième cas, nous dirons qu'il s'agit d'un ***littéral négatif***. Une formule sera alors en ***forme normale conjonctive*** si elle s'écrit comme la conjonction de disjonctions de littéraux et sera en ***forme normale disjonctive*** si elle s'écrit comme la disjonction de conjonctions de littéraux. Par exemple, la formule :

$$p_1 \wedge (p_2 \vee \neg p_3 \vee \neg p_1) \wedge (\neg p_2 \vee p_3)$$

est en forme normale conjonctive, alors que la formule :

$$p_2 \vee (\neg p_1 \wedge \neg p_3) \vee (\neg p_1 \wedge p_2) \vee (\neg p_2 \wedge p_3)$$

est en forme normale disjonctive.

On peut voir une formule en forme normale conjonctive comme une liste de disjonctions de littéraux reliées par le connecteur $\wedge$. Chacune de ces disjonctions est appelée une ***clause***. Lorsqu'une clause contient au plus un littéral positif (c'est à dire 0 ou 1 littéral positif), cette clause est une ***clause de Horn***.

Les disjonctions de littéraux et les conjonctions de littéraux peuvent être représentées par un même type regroupant dans des listes, d'une part les littéraux positifs, et d'autre part les littéraux négatifs.

```
type 'a list_lit = {lit_positif: 'a list; lit_negatif: 'a list};;
```

Une formule en forme normale sera alors représentée par une liste de disjonctions ou de conjonctions de littéraux, c'est à dire par une liste dont les éléments sont de type `'a list_lit`. Bien sûr, en adoptant cette représentation, on perd l'information permettant de savoir si les littéraux présents dans les deux listes sont reliés par un $\wedge$ ou un $\vee$. Ceci n'est pas génant puisqu'il suffira de disposer de deux fonctions différentes, correspondant aux deux cas possibles, pour transformer une formule représentée par un objet de ce type en une formule de type `'a formul`. Une clause est représentée par un objet de type `'a list_lit`.

1.3 Interprétations et modèles

Pour le moment nous n'avons pas donné de signification aux formules de la logique des propositions : nous savons seulement construire cet ensemble ou bien déterminer si une formule est syntaxiquement correcte ou non. Nous allons à présent interpréter ces formules en leur associant une valeur. Tout comme les expressions arithmétiques s'évaluent en une valeur numérique, les formules de $\mathcal{F}_p$ s'évaluent en une valeur booléenne, c'est à dire un élément de $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. Pour pouvoir associer une signification aux formules de $\mathcal{F}_p$, il faut tout d'abord associer une signification à chacun des symboles qui peuvent apparaître dans ces formules, c'est à dire aux éléments de $\Sigma_p \cup \Sigma_c$. Par exemple, pour pouvoir dire si la formule $p \Rightarrow q$ est "vraie" il faut non seulement connaître la signification du connecteur $\Rightarrow$ mais aussi connaître les "valeurs de vérité" de p et q. Alors que la signification des connecteurs logiques est fixe, celle des symboles propositionnels ne l'est pas puisque, par nature, ces symboles dénotent des variables propositionnelles qui peuvent être soit vraies, soit fausses. Il n'est donc pas possible d'évaluer une formule sans connaître la valeur de vérité des symboles propositionnels qui la composent : une formule s'évaluera donc étant donnée une interprétation de ces symboles, encore appelée une valuation. Une ***valuation*** est une fonction ν de Σ_p dans $\mathbb{B}$.

En OCAML, on pourra représenter une valuation par une liste de couples (p, b) où p est un symbole propositionnel et b un booléen. Une valuation sera donc de type `('a * bool) list`. Par exemple, la valuation ν sur $\Sigma_p = \{p_1, p_2, p_3\}$ telle que $\nu(p_1) = \mathsf{true}$, $\nu(p_2) = \mathsf{true}$ et $\nu(p_3) = \mathsf{false}$ sera représentée par la liste $[(p_1, \mathsf{true}); (p_2, \mathsf{true}); (p_3, \mathsf{false})]$. Pour connaître la valeur associée à un symbole propositionnel par une valuation représentée de cette façon, on pourra utiliser la fonction prédéfinie `assoc`. Par exemple, l'expression `(assoc` $p2$ $[(p_1, \mathsf{true}); (p_2, \mathsf{true}); (p_3, \mathsf{false})]$`)` s'évalue à la valeur `true`. Une telle représentation ne correspond pas exactement à la définition donnée puisqu'une valuation ν associe un booléen à tous les symboles propositionnels alors qu'une liste de couples de la forme (p, b) peut ne fournir la valeur associée à un symbole propositionnel que pour un sous-ensemble de Σ_p. En d'autres termes, dans la définition classique d'une valuation, ν est vue comme une fonction totale alors que sa représentation en OCAML permet de l'envisager comme une fonction partielle. Toutefois, nous verrons pas la suite que cette possibilité peut être utile dans certains cas.

Il reste maintenant à interpréter les symboles de Σ_c. Puisque l'on connait les valeurs booléennes des symboles propositionnels, ces connecteurs vont s'interpréter comme des fonctions booléennes (c'est à dire des fonctions dont les arguments et le résultat sont dans $\mathbb{B}$). Comme nous l'avons déjà dit, la signification qui est donnée à ces connecteurs ne dépend pas des valeurs associées aux symboles propositionnels par une valuation ν et on notera $[\neg]$, $[\wedge]$, $[\vee]$, $[\Rightarrow]$ et $[\Leftrightarrow]$ les fonctions booléennes correspondant aux connecteurs de Σ_c. Leur définition est classique et est rappelée dans la table de vérité suivante :

b_1	b_2	$[\neg]b_1$	$b_1[\wedge]b_2$	$b_1[\vee]b_2$	$b_1[\Rightarrow]b_2$	$b_1[\Leftrightarrow]b_2$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

Il est à présent possible de définir, étant donnée une valuation ν, la valeur $[\varphi]_\nu$ associée à une formule φ. On construit donc un prolongement, souvent noté $\overline{\nu}$, de ν aux formules : toute valuation ν admet un prolongement unique $\overline{\nu}$ à $\mathcal{F}$ défini comme suit. Soit φ une formule de $\mathcal{F}_p$ et une ν valuation.

$$\boxed{[\varphi]_\nu = \begin{cases} \nu(p) & \text{si } \varphi = p & (p \in \Sigma_p) \\ [\neg][\psi]_\nu & \text{si } \varphi = \neg\psi & (\psi \in \mathcal{F}_p) \\ [\psi_1]_\nu[\wedge][\psi_2]_\nu & \text{si } \varphi = \psi_1 \wedge \psi_2 & (\psi_1, \psi_2 \in \mathcal{F}_p) \\ [\psi_1]_\nu[\vee][\psi_2]_\nu & \text{si } \varphi = \psi_1 \vee \psi_2 & (\psi_1, \psi_2 \in \mathcal{F}_p) \\ [\psi_1]_\nu[\Rightarrow][\psi_2]_\nu & \text{si } \varphi = \psi_1 \Rightarrow \psi_2 & (\psi_1, \psi_2 \in \mathcal{F}_p) \\ [\psi_1]_\nu[\Leftrightarrow][\psi_2]_\nu & \text{si } \varphi = \psi_1 \Leftrightarrow \psi_2 & (\psi_1, \psi_2 \in \mathcal{F}_p) \end{cases}}$$

Nous venons donc, en associant une signification à chacun des symboles de $\Sigma = \Sigma_p \cup \Sigma_c$, de définir l'interprétation des formules de $\mathcal{F}_p$. Si l'on se replace dans le cadre un peu plus général ou l'on considère que $\mathcal{F}_p$ est l'ensemble défini inductivement à partir des symboles de Σ associés à une arité, on remarque que la définition du schéma d'interprétation des formules consiste à définir :

- un domaine d'interprétation, ici l'ensemble $\mathbb{B}$
- pour chaque symbole $o \in \Sigma$, une fonction $[o] : \mathbb{B}^n \to \mathbb{B}$ où n est l'arité de o

Ce procédé est classique et nous y reviendrons dans la section suivante.

Les fonctions $[\wedge]$ et $[\vee]$ sont prédéfinies dans le langage OCAML, il s'agit respectivement des primitives `&` et `or` portant sur des objets de type `bool`. Ces deux primitives permettent également d'implanter les fonctions $[\Rightarrow]$ et $[\Leftrightarrow]$ puisqu'il suffit de remarquer que l'on a les équivalences suivantes :

$$p \Leftrightarrow q \mathrel{\models\!\!\!\mid} (p \Rightarrow q) \wedge (q \Rightarrow p) \quad \text{et} \quad p \Rightarrow q \mathrel{\models\!\!\!\mid} \neg p \vee q$$

où $\mathrel{\models\!\!\!\mid}$ est une relation d'équivalence entre formules définie plus bas. D'autre part, on remarquera que la fonction $[\Leftrightarrow]$ peut être implantée en utilisant le résultat suivant :

$$[p \Leftrightarrow q]_\nu = \mathsf{true} \quad \text{si et seulement si} \quad [p]_\nu = [q]_\nu$$

Exercice 1.1

Définir en OCAML *une fonction* `eval_formul` *d'évaluation des formules de type :*

```
'a formul -> ('a * bool) list -> bool
```

On introduit à présent la notion de ***formules équivalentes*** : on dira que deux formules sont équivalentes si, quelque soit la valuation que l'on considère, ces deux formules s'évaluent à la même valeur.

$$\boxed{\forall \varphi, \psi \in \mathcal{F}_p \qquad \varphi \mathrel{\models\!\!\!\mid} \psi \text{ si et seulement si } \forall \nu,\ [\varphi]_\nu = [\psi]_\nu}$$

Par définition, cette relation est réflexive, symétrique et transitive et est donc une relation d'équivalence sur $\mathcal{F}_p$. Par exemple, il est clair que les formules $p \wedge q$ et $q \wedge p$ sont équivalentes quelles que soient les valeurs des symboles propositionnels p et q puisque l'on peut vérifier que l'opération $[\wedge]$ est commutative.

Pour déterminer si deux formules φ et ψ sont équivalentes il faut donc comparer pour toute valuation ν les valeurs $[\varphi]_\nu$ et $[\psi]_\nu$. Or si l'ensemble Σ_p est infini, l'énumération de toutes les valuations possibles est elle aussi infinie (i.e., ne termine pas). Toutefois, on peut remarquer que l'évaluation d'une formule φ dépend uniquement des valeurs associées aux symboles propositionnels qui la composent. Par exemple, pour évaluer la formule $p \wedge q$, il suffit de connaître les valeurs de p et q. Etant donné un ensemble fini E de symboles propositionnels, on définit une relation $=_E$ sur l'ensemble des valuations comme suit :

$$\nu_1 =_E \nu_2 \quad \text{si et seulement si} \quad \forall p \in E,\ \nu_1(p) = \nu_2(p)$$

Autrement dit, $\nu_1 =_E \nu_2$ si et seulement si les valuations ν_1 et ν_2 associent les mêmes valeurs à tous les éléments de E. On montre alors facilement que le résultat de l'évaluation d'une formule φ ne dépend que de la valeur associée aux symboles propositionnels qui la composent.

Soit φ une formule, E l'ensemble des symboles propositionnels qui la composent, et ν_1 et ν_2 deux valuations. Si $\nu_1 =_E \nu_2$ alors $[\varphi]_{\nu_1} = [\varphi]_{\nu_2}$.

Aussi, en notant $\nu_{|\varphi}$ la restriction d'une valuation ν aux symboles propositionnels apparaissant dans φ, on montre facilement que $[\varphi]_\nu = [\varphi]_{\nu_{|\varphi}}$. En utilisant cette propriété, il devient possible de déterminer en temps fini si deux formules sont équivalentes. En effet, même si l'ensemble Σ_p est infini, puisque l'ensemble des formules est défini de manière inductive, les formules sont obtenues en appliquant un nombre de fois fini les constructeurs et contiennent donc un nombre fini de symboles propositionnels. Pour tester l'équivalence de deux formules, il suffira donc de considérer l'ensemble des valuations portant sur l'ensemble fini des symboles propositionnels apparaissant dans les deux formules envisagées. Cet ensemble est fini et il est alors possible, en temps fini, de comparer le résultat de l'évaluation des deux formules pour toutes les valuations de cet ensemble.

Exercice 1.2

Pour tester l'équivalence de deux formules en temps fini, il faut pouvoir disposer de l'ensemble fini des symboles apparaissant dans ces formules. Nous allons utiliser les listes (sans doublons) pour représenter les ensembles.

1. *Définir en* OCAML *une fonction* `symbols_formul` *calculant l'ensemble (sans doublons) des symboles apparaissant dans une formule (le type de cette fonction est* `'a formul -> 'a list`*).*

Il nous faut à présent, étant donné un ensemble fini E de symboles propositionnels, calculer l'ensemble des valuations possibles sur E. Nous avons vu qu'une valuation était représentée par une liste. Cet ensemble sera donc une liste de listes.

2. *Définir en* OCAML *une fonction* `list_valuations` *permettant de calculer l'ensemble des valuations possibles sur un ensemble E. Le type de cette fonction est :*

```
'a list -> ('a * bool) list list
```

Pour tester l'équivalence de deux formules, il suffit maintenant de construire l'ensemble E des symboles propositionnels apparaissant dans ces deux formules, puis de construire l'ensemble des valuations sur E, et de vérifier que pour chacune de ces valuations les deux formules s'évaluent à la même valeur.

3. *Définir en* OCAML *une fonction* `equiv_formules` *permettant de tester si deux formules sont équivalentes.*

Etant donnée une valuation ν, nous savons maintenant déterminer si une formule φ est vraie lorsque la valeur de chacun de ses symboles propositionnels est déterminée par ν. Dans le cas où $[\varphi]_\nu =$ `true`, on dira que φ est ***satisfiable***, qu'elle est satisfaite par ν, ou bien que ν est un ***modèle*** de φ et on écrira $\models_\nu \varphi$. Quand il n'existe aucune valuation qui satisfait φ, φ et dite ***insatisfiable*** et si toutes les valuations satisfont φ, alors φ est appelée une ***tautologie***.

On étend ces définitions aux ensembles de formules : si Γ est un ensemble de formules, on dira qu'une valuation ν satisfait Γ, ce que l'on notera $\models_\nu \Gamma$, si et seulement si $\forall \varphi \in \Gamma, [\varphi]_\nu =$ `true` et dans ce cas on dira que ν est un modèle de Γ (c'est à dire une valuation qui satisfait simultanément toutes les formules de Γ). Quand une telle valuation existe, Γ est dit satisfiable, dans le cas contraire Γ est dit insatisfiable.

Exercice 1.3

Pour déterminer si une formule est satisfiable, il suffit de construire l'ensemble des valuations possibles portant sur l'ensemble des symboles propositionnels qui apparaissent dans cette formulle, puis de vérifier si parmi les valuations présentes dans cet ensemble, il en existe une qui satisfait la formule considérée. On procède de la même manière pour déterminer si une formule est une tautologie en vérifiant que toutes les valuations satisfont la formule considérée.

1. *Définir en* OCAML *une fonction* `satisfiable` *qui permet de tester si une formule φ est satisfiable.*

2. *Définir en* OCAML *une fonction* `tautologie` *qui permet de tester si une formule φ est une tautologie.*

L'ensemble des valuations possibles sur les symboles propositionnels apparaissant dans une formule φ contenant n symboles propositionnels distincts contient 2^n éléments. Aussi, déterminer si une formule est satisfiable ou est une tautologie peut conduire à envisager 2^n valuations ce qui est tout à fait inefficace lorsque n est grand. Nous présenterons par la suite des méthodes plus efficaces, dans certains cas, permettant de répondre à ces questions.

Bien souvent, il peut être utile de déterminer si une formule est vraie à chaque fois qu'une ou plusieurs autres formules sont vraies. C'est le cas par exemple lorsque l'on veut établir qu'une formule est vraie en faisant pour hypothèse qu'une ou plusieurs autres formules sont vraies. On introduit donc la notion de conséquence sémantique : une formule φ est une ***conséquence***

sémantique d'une autre formule ψ si chaque valuation qui satisfait ψ satisfait aussi φ. Dans ce cas on écrira $\psi \models \varphi$. Par exemple, on vérifie facilement que la formule p est une conséquence sémantique de la formule $p \wedge q$ puisque les seules valuations qui satisfont $p \wedge q$ sont les valuations ν telles que $\nu(p) = \texttt{true}$ qui satisfont évidemment la formule p. On peut étendre la relation de conséquence sémantique de manière à envisager les valuations qui satisfont simultanément toutes les formules d'un ensemble de formules. On dira alors qu'une formule φ est conséquence sémantique d'un ensemble Γ de formules si chaque valuation qui satisfait simultanément les formules de Γ satisfait aussi φ. Dans ce cas, par abus de notation, on écrira $\Gamma \models \varphi$.

$$\boxed{\begin{array}{lll} \psi \models \varphi & \text{si et seulement si} & \forall \nu,\ (\text{si } \models_\nu \psi \text{ alors} \models_\nu \varphi) \\ \Gamma \models \varphi & \text{si et seulement si} & \forall \nu,\ (\text{si } (\forall \psi \in \Gamma\ \models_\nu \psi) \text{ alors} \models_\nu \varphi) \end{array}}$$

Par exemple, on peut facilement vérifier que $\{p, q\} \models p \wedge q$ puisque les seules valuations qui satisfont simultanément les formules p et q sont les valuations ν telles que $\nu(p) = \texttt{true}$ et $\nu(q) = \texttt{true}$ qui satisfont évidemment la formule $p \wedge q$.

Exercice 1.4

Pour déterminer si une formule φ est conséquence sémantique d'une formule ψ, il faut tout d'abord construire l'ensemble des valuations possibles portant sur les symboles apparaissant dans ψ ou φ, puis sélectionner celles qui satisfont ψ.

1. *Définir en* OCAML *une fonction* `list_valuations_true` *permettant d'obtenir à partir d'une liste de valuations la liste de celles qui satisfont une formule.*

2. *En déduire une fonction* `cons_sem` *qui permet de déterminer si une formule φ est conséquence sémantique d'une formule ψ.*

On procède de manière identique pour déterminer si une formule φ est conséquence sémantique d'une ensemble Γ de formules. On doit donc construire la listes des valuations possibles sur les symboles propositionnels apparaissant dans les formules de Γ ou φ, puis sélectionner parmi elles celles qui satisfont simultanément toutes les formules de Γ :

3. *Définir en* OCAML *une fonction* `cons_sem_E` *qui permet de déterminer si une formule φ est conséquence sémantique d'un ensemble E de formules.*

Remarque. Il est possible de redéfinir l'équivalence “logique” entre deux formules en utilisant la notion de conséquence sémantique. En effet, on peut facilement montrer que deux formules φ et ψ sont équivalentes si et seulement si φ est conséquence sémantique de ψ et *vice-versa* :

$$\varphi \models\!\!\!\!\dashv \psi \text{ si et seulement si } (\varphi \models \psi \text{ et } \psi \models \varphi)$$

1.4 Mise en forme normale

Nous présentons ici deux méthodes permettant d'obtenir, à partir d'une formule quelconque φ, une formule ψ en forme normale (conjonctive ou disjonctive) telle que $\varphi \models\!\!\!\!\dashv \psi$. En fait, ces deux méthodes constituent une démonstration constructive de la proposition suivante :

> Pour toute formule $\varphi \in \mathcal{F}_p$, il existe une formule ψ_1 en forme normale conjonctive et une formule ψ_2 en forme normale disjonctive telles que $\varphi \models\!\!\!\!\dashv \psi_1 \models\!\!\!\!\dashv \psi_2$.

En effet, la démonstration de l'existence des formules ψ_1 et ψ_2 est constructive puisque l'on indique ici un algorithme permettant de construire effectivement ces deux formules. Il restera donc à montrer que ces algorithmes terminent et préservent l'équivalence, ce que nous ferons pour la première méthode.

Dans ce qui suit, l'emploi des termes "forme normale" et "normalisation" peut sembler abusif. En effet, une même formule admet plusieurs formes normales conjonctives ou disjonctives équivalentes (par exemple $p \wedge (q \vee r)$ et $(r \vee q) \wedge p$ sont deux formules en forme normale conjonctive équivalentes). Il n'y a donc pas unicité[1].

1.4.1 Mise en forme normale par transformations syntaxiques

Il est possible d'obtenir la forme normale d'une formule par transformations successives en lui appliquant les "règles de réécriture" suivantes dans l'ordre indiqué. Ces règles reposent sur des équivalences entre formules qui s'établissent facilement. A l'issue de chacune des étapes décrites ci-dessous, l'équivalence entre les formules de départ et les formules d'arrivée est donc bien garantie.

(1.) Elimination des connecteurs $\Rightarrow$ et $\Leftrightarrow$ Une formule en forme normale ne contient plus que les connecteurs $\neg$, $\wedge$ et $\vee$. Il s'agit donc tout d'abord d'éliminer toutes les occurrences des deux connecteurs $\Rightarrow$ et $\Leftrightarrow$ de la formule considérée. Pour ce faire, on utilise les deux équivalences suivantes :

$$(\varphi \Leftrightarrow \psi) \models\!\mid ((\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)) \qquad (\varphi \Rightarrow \psi) \models\!\mid (\neg\varphi \vee \psi)$$

vues comme des règles de réécriture (de la gauche vers la droite). On éliminera tout d'abord le connecteur $\Leftrightarrow$ puis le connecteur $\Rightarrow$. A chaque fois que l'on applique ces transformations, une occurrence du connecteur considéré disparait. Le nombre d'occurrences de ce connecteur étant fini, cette étape termine.

Exercice 1.5

Définir en OCAML *une fonction* `elim_impl_equiv` *qui permet, de manière récursive, d'éliminer simultanément toutes les occurrences des connecteurs* $\Leftrightarrow$ *et* $\Rightarrow$ *d'une formule.*

(2.) Déplacement des connecteurs de négation Dans une formule en forme normale, les négations s'appliquent uniquement sur des symboles propositionnels. On cherche donc à présent à déplacer "le plus à l'intérieur possible des formules" le connecteur $\neg$. Pour cela, on utilise les deux équivalences suivantes :

$$\neg(\varphi \wedge \psi) \models\!\mid (\neg\varphi \vee \neg\psi) \qquad \neg(\varphi \vee \psi) \models\!\mid (\neg\varphi \wedge \neg\psi)$$

Il reste alors à simplifier la formule obtenue en considérant l'équivalence :

$$\neg\neg\varphi \models\!\mid \varphi$$

Ici encore, l'équivalence des formules obtenues après cette transformation est garantie et cette étape termine puisqu'à chaque itération, la profondeur d'un des connecteurs augmente strictement, cette profondeur étant bornée puisque les formules sont finies.

Exercice 1.6

Définir en OCAML *une fonction* `neg_prop` *qui permet, de manière récursive, d'appliquer l'étape (2.) à partir d'une formule ne contenant plus d'occurrences des connecteurs* $\Leftrightarrow$ *et* $\Rightarrow$*. Afin de prendre en compte la simplification qui transforme* $\neg\neg\varphi$ *en* φ*, un filtrage supplémentaire pourra être effectué dans le cas où la formule considérée a été construite à partir du connecteur* $\neg$*.*

[1] comme c'est le cas, par exemple, pour les termes du λ-calcul qui se normalisent de façon unique

(3.) Distributivité A l'issue des deux étapes précédentes, la formule considérée ne comporte plus que des littéraux (positifs ou négatifs) reliés entre eux par les connecteurs $\wedge$ et $\vee$. Nous allons donc utiliser les propriétés de distributivité de ces opérateurs pour obtenir la formule en forme normale. Cette trannsformation repose sur les équivalences suivantes :

$$(\varphi \wedge (\psi \vee \phi)) \models\!\mid ((\varphi \wedge \psi) \vee (\varphi \wedge \phi)) \qquad (\varphi \vee (\psi \wedge \phi)) \models\!\mid ((\varphi \vee \psi) \wedge (\varphi \vee \phi))$$

La distributivité de $\wedge$ sur $\vee$ va permettre d'obtenir la formule en forme normale disjonctive alors que la distributivité de $\vee$ sur $\wedge$ permettra d'obtenir cette formule en forme normale conjonctive. Ici encore, l'équivalence entre les formules est préservée par cette transformation qu'il suffit d'appliquer un nombre de fois fini pour obtenir une forme normale.

Exercice 1.7

Pour implanter l'étape (3.), il suffit de procéder de manière récursive. On remarquera que lorsque la formule considérée a été obtenue à partir du connecteur $\neg$, celui-ci porte nécessairement sur un symbole propositionnel puisque l'étape (2.) a été appliquée précédemment. Enfin, un test pourra être réalisé pour détecter lorsque la formule est en forme normale et dans ce cas aucun appel récursif n'est effectué. Définir en OCAML *deux fonctions* `distrib_and` *et* `distrib_or` *qui appliquent respectivement la distributivité de $\wedge$ sur $\vee$ et de $\vee$ sur $\wedge$ jusqu'à obtenir un point fixe (c'est à dire jusqu'à ce que la propriété de distributivité ne puisse plus être appliquée).*

(4.) Simplifications A l'issue de ces trois étapes, les formules obtenues sont en forme normale. On peut toutefois les simplifier. En effet, une formule en forme normale conjonctive (resp. disjonctive) s'écrit :

$$\psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_n \qquad (\text{resp.} \quad \psi_1 \vee \psi_2 \vee \cdots \vee \psi_n)$$

où les formules ψ_i sont des disjonctions (resp. des conjonctions) de littéraux. Deux situations peuvent conduire à simplifier la formule.

- Si un littéral apparaît plusieurs fois dans une disjonction (resp. dans une conjonction), on peut n'en conserver qu'une unique occurrence puisque l'on a l'équivalence $p \vee p \models\!\mid p$ (resp. $p \wedge p \models\!\mid p$).
- Si dans une disjonction (resp. conjonction) il existe des littéraux opposés, c'est à dire le littéral p et le littéral $\neg p$, alors cette disjonction (resp. conjonction) est une tautologie (resp. insatisfiable) puisque pour toute valuation ν, $[p \vee \neg p]_\nu = \mathsf{true}$ (resp. ν, $[p \wedge \neg p]_\nu = \mathsf{false}$). Les formules ψ_i étant reliées par le connecteur $\wedge$ (resp. $\vee$), on peut alors supprimer de la formule en forme normale cette disjonction (resp. cette conjonction) puisque, pour tout booléen b, on a $b[\wedge]\mathsf{true} = b$ (resp. $b[\vee]\mathsf{false} = b$).

Exercice 1.8

Pour implanter l'étape (4.), nous allons tout d'abord transformer la représentation de la forme normale obtenue à l'étape précédente en utilisant le type `list_lit` *déjà défini. En effet, détecter la présence de littéraux opposés au sein d'une disjonction (resp. d'une conjonction) sera plus aisé si cette disjonction (resp. cette conjonction) est représentée sous forme de listes.*

1. *Définir en* OCAML *deux fonctions* `flat_or` *et* `flat_and` *permettant d'obtenir un objet de type* `list_lit` *à partir d'une disjonction et d'une conjonction de littéraux. Ces deux fonctions utiliseront une fonction* `fusion_normal_form` *qui permet de fusionner deux objets de type* `list_lit` *en supprimant les doublons.*

2. *Utiliser ces fonctions pour définir deux fonctions* `transfo_FNC` *et* `transfo_FND` *permettant de transformer une formule de type* `'a formul` *en une liste d'objets de type* `list_lit`*.*

3. *Il reste maintenant à effectuer les simplifications portant sur la présence de littéraux opposés dans les disjonctions et les conjonctions. Définir deux fonctions* `litt_oppos` *et* `sup_litt_oppos` *qui effectuent cette simplification (la première détecte la présence de littéraux opposés et la deuxième supprime de la liste les objets contenant des littéraux opposés).*

Exercice 1.9

Déduire des fonctions définies dans les exercices précédents deux fonctions `fnc` *et* `fnd` *de mise en forme normale conjonctive et disjonctive.*

Exemple En utilisant cette méthode, la formule $\neg(p_1 \Leftrightarrow (p_2 \Rightarrow p_3))$ est transformée comme suit :

$$
\begin{array}{ll}
 & \neg(p_1 \boxed{\Leftrightarrow} (p_2 \Rightarrow p_3)) \\
(1.) & \neg((\neg p_1 \vee (p_2 \boxed{\Rightarrow} p_3)) \wedge (p_1 \vee \neg(p_2 \boxed{\Rightarrow} p_3))) \\
(1.) & \boxed{\neg}((\neg p_1 \vee (\neg p_2 \vee p_3)) \wedge (p_1 \vee \neg(\neg p_2 \vee p_3))) \\
(2.) & (\boxed{\neg}(\neg p_1 \vee (\neg p_2 \vee p_3)) \vee \boxed{\neg}(p_1 \vee \neg(\neg p_2 \vee p_3))) \\
(2.) & ((\neg\neg p_1 \wedge \boxed{\neg}(\neg p_2 \vee p_3)) \vee (\neg p_1 \wedge \neg\neg(\neg p_2 \vee p_3))) \\
(2.) & ((\boxed{\neg\neg} p_1 \wedge (\boxed{\neg\neg} p_2 \wedge \neg p_3)) \vee (\neg p_1 \wedge \boxed{\neg\neg}(\neg p_2 \vee p_3))) \\
(2.) & ((p_1 \wedge (p_2 \wedge \neg p_3)) \vee (\neg p_1 \wedge (\neg p_2 \vee p_3)))
\end{array}
$$

L'étape (3.) permet d'obtenir directement la forme normale disjonctive :

$$
\begin{array}{l}
((p_1 \wedge (p_2 \wedge \neg p_3)) \vee (\neg p_1 \boxed{\wedge} (\neg p_2 \underline{\vee} p_3))) \\
(p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_3)
\end{array}
$$

Pour obtenir la forme normale conjonctive on applique l'étape (3.) de manière différente :

$$
\begin{array}{ll}
 & ((p_1 \wedge (p_2 \wedge \neg p_3)) \boxed{\vee} (\neg p_1 \underline{\wedge} (\neg p_2 \vee p_3))) \\
(3.) & ((p_1 \underline{\wedge} p_2 \underline{\wedge} \neg p_3) \boxed{\vee} \neg p_1) \wedge ((p_1 \underline{\wedge} p_2 \underline{\wedge} \neg p_3) \boxed{\vee} (\neg p_2 \vee p_3)) \\
(3.) & (p_1 \vee \neg p_1) \wedge (p_2 \vee \neg p_1) \wedge (\neg p_3 \vee \neg p_1) \wedge (p_1 \vee \neg p_2 \vee p_3) \\
 & \wedge (p_2 \vee \neg p_2 \vee p_3) \wedge (\neg p_3 \vee \neg p_2 \vee p_3) \\
(4.) & (p_2 \vee \neg p_1) \wedge (\neg p_3 \vee \neg p_1) \wedge (p_1 \vee \neg p_2 \vee p_3)
\end{array}
$$

1.4.2 Construction d'une forme normale basée sur les valuations

Il est possible d'obtenir une forme normale d'une formule φ en considérant l'ensemble des valuations qui satisfont cette formule. Considérons par exemple la construction d'une forme normale disjonctive ψ d'une formule φ qui peut s'obtenir comme suit[2] :

1. On détermine tout d'abord les deux ensembles suivants : l'ensemble P des symboles propositionnels apparaissant dans φ et l'ensemble E des valuations ν restreintes aux symboles de P, notées $\nu_{|\varphi}$, qui satisfont φ. Ces deux ensembles sont finis.
$$P = \{p_1, \cdots, p_n\} \qquad E = \{\nu_{|\varphi} \mid [\varphi]_\nu = \mathsf{true}\}$$

2. A chaque valuation $\nu_j \in E$ on associe la formule ψ_j :
$$e_1 p_1 \wedge \cdots \wedge e_n p_n$$
où pour tout i $(1 \leq i \leq n)$, $e_i p_i$ est la formule p_i si $[p_i]_{\nu_j} = \mathsf{true}$ et $\neg p_i$ sinon.

[2]Une méthode similaire permet de déterminer la forme normale conjonctive d'une formule.

3. ψ est la formule $\psi_1 \vee \cdots \vee \psi_j$.

Par exemple, en utilisant cette méthode, la forme normale disjonctive de la formule $\neg(p_1 \Leftrightarrow (p_2 \Rightarrow p_3))$ est obtenue commme suit. Quatre valuations, ν_1, ν_2, ν_3 et ν_4, restreintes à l'ensemble de symboles propositionnels $\{p_1, p_2, p_3\}$ satisfont la formule considérée. On associe à ces quatre valuations les quatre formules ψ_1, ψ_2, ψ_3 et ψ_4.

$$\begin{array}{lll|ccccccc}
\nu_1(p_1) = \mathsf{true} & \nu_1(p_2) = \mathsf{true} & \nu_1(p_3) = \mathsf{false} & p_1 & \wedge & p_2 & \wedge & \neg p_3 & (\psi_1) \\
\nu_2(p_1) = \mathsf{false} & \nu_2(p_2) = \mathsf{true} & \nu_2(p_3) = \mathsf{true} & \neg p_1 & \wedge & p_2 & \wedge & p_3 & (\psi_2) \\
\nu_3(p_1) = \mathsf{false} & \nu_3(p_2) = \mathsf{false} & \nu_3(p_3) = \mathsf{true} & \neg p_1 & \wedge & \neg p_2 & \wedge & p_3 & (\psi_3) \\
\nu_4(p_1) = \mathsf{false} & \nu_4(p_2) = \mathsf{false} & \nu_4(p_3) = \mathsf{false} & \neg p_1 & \wedge & \neg p_2 & \wedge & \neg p_3 & (\psi_4)
\end{array}$$

La forme normale disjonctive ainsi construite est donc :

$$(p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge p_2 \wedge p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_3)$$

Exercice 1.10

Définir en OCAML *une fonction* `fnd2` *qui implante cette construction (en utilisant la fonction* `list_valuations_true` *définie dans l'exercice 1.4, il suffit de construire la liste des valuations qui satisfont la formule φ considérée puis de transformer chacune de ces valuations en une conjonction de littéraux représentée par un objet de type* `list_lit`*).*

Remarque. On peut remarquer que mettre une formule φ sous forme normale conjonctive (resp. disjonctive) revient à mettre la formule $\neg\varphi$ sous forme normale disjonctive (resp. conjonctive). En effet, soit $(l_1^1 \wedge \cdots \wedge l_{n_1}^1) \vee \cdots \vee (l_1^k \wedge \cdots \wedge l_{n_k}^k)$ la forme normale disjonctive de $\neg\varphi$. On a :

$$\begin{array}{rl}
 & \neg((l_1^1 \wedge \cdots \wedge l_{n_1}^1) \vee \cdots \vee (l_1^k \wedge \cdots \wedge l_{n_k}^k)) \\
\models\!\mid & \neg(l_1^1 \wedge \cdots \wedge l_{n_1}^1) \wedge \cdots \wedge \neg(l_1^k \wedge \cdots \wedge l_{n_k}^k) \\
\models\!\mid & (\neg l_1^1 \vee \cdots \vee \neg l_{n_1}^1) \wedge \cdots \wedge (\neg l_1^k \vee \cdots \vee \neg l_{n_k}^k)
\end{array}$$

Par exemple, pour obtenir la forme normale conjonctive de la formule $(p_1 \Leftrightarrow (p_2 \Rightarrow p_3))$, on peut construire la forme normale disjonctive de $\neg(p_1 \Leftrightarrow (p_2 \Rightarrow p_3))$ qui est $(p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_3)$, puis considérer la négation de cette formule en appliquant les équivalences utilisées précédemment :

$$\begin{array}{rl}
 & \neg((p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_3)) \\
\models\!\mid & \neg(p_1 \wedge p_2 \wedge \neg p_3) \wedge \neg(\neg p_1 \wedge \neg p_2) \wedge \neg(\neg p_1 \wedge p_3) \\
\models\!\mid & (\neg p_1 \vee \neg p_2 \vee \neg\neg p_3) \wedge (\neg\neg p_1 \vee \neg\neg p_2) \wedge (\neg\neg p_1 \vee \neg p_3) \\
\models\!\mid & (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_2) \wedge (p_1 \vee \neg p_3)
\end{array}$$

On obtient ainsi la forme normale conjonctive de la formule $\neg(p_1 \Leftrightarrow (p_2 \Rightarrow p_3))$.

1.5 Arbres sémantiques

Nous allons à présent utiliser la proposition établissant pour toute formule φ l'existence d'une formule ψ en forme normale conjonctive telle que $\psi \models\!\mid \varphi$, pour tenter de déterminer si une formule φ est vraie de manière plus efficace qu'en énumérant toutes les valuations possibles sur les symboles propositionnels qui la composent. Pour montrer qu'une formule φ est une tautologie, il suffit de montrer qu'il n'existe pas de valuation qui satisfait $\neg\varphi$. On utilise donc la proposition suivante qui se prouve facilement.

Pour toute formule φ, $\models \varphi$ si et seulement si $\neg\varphi$ est insatisfiable.

La méthode des arbres sémantiques permet de montrer qu'une formule ψ est insatisfiable lorsque ψ est en forme normale conjonctive. Aussi, pour établir qu'une formule φ est vraie, il suffit de considérer la forme normale conjonctive ψ de $\neg\varphi$, puis de construire l'arbre sémantique associé à ψ et défini comme suit.

> Soit ψ une formule en forme normale conjonctive et e une énumération des symboles propositionnels qui la composent.
>
> 1. L'arbre sémantique $\mathcal{A}[\psi, e]$ associé à ψ relativement à l'énumération e est un arbre binaire dont les noeuds sont étiquetés par des valuations partielles (i.e., portant sur un sous-ensemble fini de Σ_p) tel que la racine est étiquetée par la valuation définie "nulle part" et tel que le fils gauche (resp. droit) n d'un noeud n', de profondeur i, étiqueté par la valuation ν', est étiqueté par la valuation ν définie par :
>
> $$\nu(q) = \begin{cases} \mathsf{true} \text{ (resp. } \mathsf{false}\text{)} & \text{si } q = p \\ \nu'(q) & \text{sinon} \end{cases}$$
>
>   où p est le i-ième élément de e.
>
> 2. Une feuille de l'arbre sémantique $\mathcal{A}[\psi, e]$ est dite fermée si elle est étiquetée par une valuation partielle ν telle que quel que soit le prolongement ν_p de ν, $[\psi]_{\nu_p} = \mathsf{false}$.
>
> 3. Un arbre sémantique est fermé si toutes ses feuilles sont fermées.

On montre alors facilement la proposition suivante :

> Une formule φ est insatisfiable si elle peut être associée à un arbre sémantique fermé.

Considérons par exemple la formule φ suivante :

$$\neg r \wedge (q \vee \neg p) \wedge (r \vee \neg p \vee \neg q) \wedge p$$

et l'énumération $e = \langle r, q, p \rangle$. La racine n_0 de cet arbre est par définition étiquetée par la valuation ν_0 définie nulle part.

- Le fils gauche n_1 de la racine est étiqueté par la valuation ν_1 telle que $\nu_1(r) = \mathsf{true}$. Il s'agit d'une feuille fermée puisque quel que soit le prolongement ν_q de ν_1 on aura $[\varphi]_{\nu_q} = \mathsf{false}$. En effet, une des disjonctions présentes dans φ contient l'unique littéral $\neg r$ qui ne sera jamais satisfait par un prolongement de ν_1 et puisque les disjonctions sont reliées par le connecteur $\wedge$, la conjonction des disjonctions ne pourra pas être satisfaite par une valuation qui associe true au symbole propositionnel r.

- Le fils droit n_2 de la racine est étiqueté par la valuation ν_2 telle que $\nu_2(r) = \mathsf{false}$. Il ne s'agit pas d'une feuille fermée puisque l'on ne sait pas encore s'il existe des prolongements de ν_2 qui satisfont φ. Par contre, on sait que s'ils existent, ces prolongements satisfont la formule $\neg r \wedge (q \vee \neg p) \wedge (\neg p \vee \neg q) \wedge p$. En effet, puisque $\nu_2(r) = \mathsf{false}$, on peut supprimer de la disjonction $r \vee \neg p \vee \neg q$ le littéral r puisque ce dernier ne pourra jamais être satisfait.

  - Le fils gauche n_{21} du noeud n_2 est étiqueté par la valuation ν_{21} telle que $\nu_{21}(r) = \mathsf{false}$ et $\nu_{21}(q) = \mathsf{true}$. Ici encore, on ne sait pas s'il existe un prolongement de ν_{21} qui satisfait φ. Par contre, on sait que si un tel prolongement existe, il satisfait la formule $\neg r \wedge (q \vee \neg p) \wedge \neg p \wedge p$. En effet, puisque $\nu_{21}(q) = \mathsf{true}$, on supprime des disjonctions présentes dans la formule obtenue à l'étape précédente les occurrences du littéral $\neg q$.

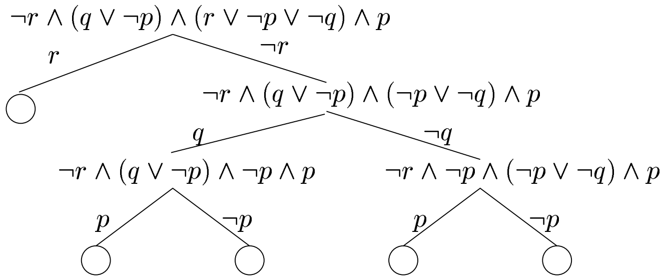

Figure 1: Un arbre sémantique pour la formule $\neg r \wedge (q \vee \neg p) \wedge (r \vee \neg p \vee \neg q) \wedge p$

Les fils gauche et droit de ce noeud correspondront alors à des feuilles fermées puisque deux des disjonctions contiennent un unique littéral qui est p et $\neg p$. Quel que soit le prolongement (à p) de ν_{21}, la formule φ ne sera pas satisfaite.

– Le fils droit n_{22} du noeud n_2 est étiqueté par la valuation ν_{22} telle que $\nu_{22}(r) = \mathsf{false}$ et $\nu_{22}(q) = \mathsf{false}$. Ici encore, on ne sait pas s'il existe un prolongement de ν_{22} qui satisfait φ mais on sait que si un tel prolongement existe, il satisfait la formule $\neg r \wedge \neg p \wedge (\neg p \vee \neg q) \wedge p$. En effet, puisque $\nu_{22}(q) = \mathsf{false}$, on supprime des disjonctions présentes dans la formule obtenue à l'étape précédente les occurrences du littéral q.

Les fils gauche et droit des noeuds n_{21} et n_{22} correspondront alors à des feuilles fermées pour les mêmes raisons que dans le cas précédent.

Puisque l'arbre sémantique $\mathcal{A}[\varphi, e]$ que nous venons de décrire est fermé, la formule φ est insatisfiable. Chaque chemin de cet arbre correspond à la construction d'une valuation partielle. Sur cet exemple, l'utilisation d'un arbre sémantique permet de ne considérer que 5 des 8 valuations possibles sur l'ensemble des symboles propositionnels qui apparaissent dans φ. Il est souvent pratique de représenter cet arbre en étiquetant les noeuds par les formules que l'on cherche à falsifier et en faisant figurer les littéraux considérés vrais sur les arcs. La valuation partielle associée à chaque noeud n est alors définie par les littéraux qui figurent sur les arcs du chemin menant de la racine à n. La figure 1 illustre l'arbre contruit dans l'exemple précédent. Toutefois l'ordre d'énumeration de ces symboles conditionne la forme de l'arbre, qui dans le pire des cas, peut conduire à considérer toutes les valuations possibles.

Exercice 1.11

En Ocaml, *le type des arbres sémantiques peut être défini par :*

```
type 'a arbre_semantique =
    Arbre_Vide of ('a * bool) list
  | Noeud of   ('a list) * (('a list_it) list) * (('a * bool) list)
             * 'a arbre_semantique * 'a arbre_semantique ;;
```

Ici, nous enrichissons le type des étiquettes de manière à disposer pour chaque noeud de l'énumeration des symboles propositionnels restant à considérer, de la formule en forme normale conjonctive (représentée par un objet de type `('a list_it) list`*) correspondant à une version simplifiée de la formule initiale (i.e., dans laquelle on a supprimé les littéraux qui ne pourront plus être satisfaits par la valuation en cours de construction) et la valuation en cours de construction (représentée par un objet de type* `('a * bool) list`*). Par contre, les feuilles sont étiquetées uniquement par des valuations (puisqu'aucune construction supplémentaire n'aura lieu).*

1. *A chaque étape, pour obtenir une version simplifiée de la formule, on supprime de la formule les occurrences positives ou négatives d'un littéral donné (celui pour lequel on construit un prolongement). Définir une fonction* `mise_a_jour_disj` *de mise à jour d'une formule en forme normale conjonctive qui permet de supprimer un littéral en fonction de la valeur booléenne qui lui est associée par la nouvelle valuation construite. Cette fonction calcule en fait une paire dont la première composante est la nouvelle formule et dont la deuxième composante est un booléen indiquant si dans cette nouvelle formule une des disjonctions est vide. En effet, on pourra par la suite utiliser cette information pour détecter si un noeud correspond à une feuille fermée. Cette fonction pourra utiliser une fonction* `mise_a_jour_disj` *qui effectue la simplification au sein d'une disjonction.*

2. *Définir une fonction* `fils` *qui, étant donnés une formule en forme normale conjonctive, une valuation, un symbole propositionnel et la valeur booléenne qui va lui être associée par la valuation qui va être construite, calcule la nouvelle formule simplifiée et la nouvelle valuation ainsi qu'un booléen déterminant si lors de la simplification on a obtenu une disjonction vide.*

3. *Définir une fonction* `cons_arbre` *qui permet la construction de l'arbre de manière récursive. A chaque étape, il faudra tester si on est arrivé à une feuille fermée et dans ce cas arrêter la construction de la branche considérée. Cette fonction calcule en fait une paire dont la première composante est l'arbre sémantique construit et dont la deuxième composante est un booléen indiquant si l'arbre obtenu est fermé ou non.*

4. *Déduire des fonctions définies dans les questions précédentes une fonction* `tautologie2` *permettant de déterminer si une formule est une tautologie. Pour cela, on commence tout d'abord par calculer l'ensemble des symboles propositionnels qui la composent, puis on transforme la négation de cette formule en forme normale conjonctive et on teste ensuite si l'arbre sémantique que l'on peut construire à partir de cette forme normale conjonctive et de la liste des symboles propositionnels est fermé.*

Remarque. L'utilisation d'arbres sémantiques permet aussi de déterminer si une formule φ est conséquence sémantique d'un ensemble de formules Γ. En effet, il suffit pour cela de montrer que la formule :

$$\bigwedge_{\psi \in \Gamma} \psi \Rightarrow \varphi$$

est une tautologie. Mais on peut aussi procéder directement en montrant que l'ensemble $\Gamma \cup \{\neg\varphi\}$ est insatisfiable, c'est à dire qu'il n'existe pas de valuation qui satisfait simultanément toutes les formules de cet ensemble. Pour cela, il suffira de considérer la conjonction de la conjonction des formules de Γ mises en formes normales conjonctives avec la forme normale conjonctive de $\neg\varphi$. Une telle formule est encore en forme normale conjonctive et il suffit alors de vérifier qu'on peut lui associer un arbre sémantique fermé.

1.6 Preuves en logique des propositions : Résolution

A partir de la forme normale conjonctive d'une formule :

$$\begin{array}{ll} & \underbrace{(\neg a_1^1 \vee \cdots \vee \neg a_{n_1}^1 \vee b_1^1 \vee \cdots \vee b_{m_1}^1)}_{C_1} \\ \wedge & \cdots \\ \wedge & \underbrace{(\neg a_1^k \vee \cdots \vee \neg a_{n_k}^k \vee b_1^k \vee \cdots \vee b_{m_k}^k)}_{C_k} \end{array}$$

il est possible d'obtenir un ensemble fini $\{C_1, \cdots, C_k\}$ de clauses de la forme :

$$\neg a_1 \vee \cdots \vee \neg a_n \vee b_1 \vee \cdots \vee b_m$$

où les a_i et les b_i sont des symboles propositionnels. Si n et m sont strictement positifs, une telle clause peut s'écrire :

$$(a_1 \wedge \cdots \wedge a_n) \Rightarrow (b_1 \vee \cdots \vee b_m)$$

Dans ce qui suit, on désignera ces clauses par $\Gamma \triangleright \Delta$: Γ est l'ensemble $\{a_1, \cdots, a_n\}$ des symboles propositionnels qui apparaissent négativement dans la clause et Δ est l'ensemble $\{b_1, \cdots, b_m\}$ des symboles propositionnels qui apparaissent positivement dans la clause. $\Gamma \triangleright \Delta$ est un séquent qui s'interprète par : "si les formules de Γ sont vraies, alors au moins une formule de Δ est vraie" (cette notion de séquent est plus générale que celle vue en cours puisque Δ n'est pas forcément un singleton). Une clause est donc un séquent sur des formules atomiques (un atome est un littéral positif, c'est à dire un symbole propositionnel). Un tel objet permet de "faire" de la logique sans symbole logique en considérant un raisonnement combinatoire sur les symboles propositionnels.

L'intérêt des clauses provient du fait qu'en se restreignant à ne manipuler que des clauses on peut considérer un système d'inférence valide et complet qui ne contient qu'une seule règle de déduction. Il s'agit de la "***règle de coupure***" (le mot "coupure" n'a rien à voir avec le `Cut` de Prolog qui n'a lui même rien à voir avec la logique) qui permet à partir de deux clauses de déduire une clause.

$$\boxed{(Cut \quad \text{si } p \in \Delta_1 \cap \Gamma_2) : \frac{\Gamma_1 \triangleright \Delta_1 \qquad \Gamma_2 \triangleright \Delta_2}{\Gamma_1 \cup (\Gamma_2 \backslash \{p\}) \triangleright (\Delta_1 \backslash \{p\}) \cup \Delta_2}}$$

En d'autres termes, cette règle exprime simplement la déduction suivante :

$$(Cut \quad \text{si } a'_i = b_j) : \frac{\begin{array}{c} \neg a_1 \vee \cdots \vee \neg a_{n_1} \vee b_1 \vee \cdots \vee b_j \vee \cdots \vee b_{m_1} \\ \neg a'_1 \vee \cdots \vee \neg a'_i \vee \cdots \vee \neg a'_{n_2} \vee b'_1 \vee \cdots \vee b'_{m_2} \end{array}}{\begin{array}{ll} & \neg a_1 \vee \cdots \vee \neg a_{n_1} \vee \neg a'_1 \vee \cdots \vee \neg a'_{i-1} \vee \neg a'_{i+1} \vee \cdots \vee \neg a'_{n_2} \\ \vee & b_1 \vee \cdots \vee b_{j-1} \vee b_{j+1} \vee \cdots \vee b_{m_1} \vee b'_1 \vee \cdots \vee b'_{m_2} \end{array}}$$

Par exemple, on peut déduire des deux clauses $p_1 \vee p_2 \vee p_3$ et $\neg p_3 \vee p_4 \vee p_5$ par coupure "sur p_3" la clause $p_1 \vee p_2 \vee p_4 \vee p_5$. Une ***preuve par coupure*** est une suite finie de clauses obtenues par applications successives de la règle de coupure. Plus précisément, étant donné un ensemble S de clauses, une preuve de C par coupure à partir de S est une suite finie de clauses C_1, C_2, ..., C_k où C_k est la clause C et pour tout i $(1 \leq i \leq k)$:

- soit $C_i \in S$

- soit il existe deux entiers j_1 et j_2 strictement inférieurs à i tels que C_i se déduise de C_{j_1} et C_{j_2} par coupure.

Une ***réfutation par coupure*** de S est une preuve de la clause vide $\emptyset \triangleright \emptyset$ à partir de S. Par exemple, il existe une preuve de r à partir de $S = \{\{p\} \triangleright \{q,r\} ; \{p,q\} \triangleright \emptyset ; \emptyset \triangleright \{p\}\}$:

$$\begin{array}{ll} C_1 & \{p\} \triangleright \{q,r\} \\ C_2 & \{p,q\} \triangleright \emptyset \\ C_3 & \{p\} \triangleright \{r\} \\ C_4 & \emptyset \triangleright \{p\} \\ C_5 & \emptyset \triangleright \{r\} \end{array}$$

Et donc, il existe une réfutation par coupure de $S \cup \{\{r\} \triangleright \emptyset\}$.

1.7 Clauses de Horn : "miniProlog" propositionnel

Dans ce qui suit nous allons voir que la restriction de la méthode de résolution à une classe particulière de clauses, les clauses de Horn, permet d'envisager un langage (il s'agit d'un "miniProlog" propositionnel) auquel nous pouvons associer diverses sémantiques.

Dans ce qui suit, on dira qu'une clause $\Gamma \triangleright \Delta$ est :

- une ***clause de Horn*** si Δ contient au plus un élément
- une ***clause positive*** si $\Gamma = \emptyset$
- une ***clause négative*** si $\Delta = \emptyset$
- une ***clause définie*** si Δ est un singleton.

clause définie		clause négative	
	$\neg a_1 \vee \cdots \vee \neg a_n \vee b$		$\neg a_1 \vee \cdots \vee \neg a_n$
$\equiv$	$(a_1 \wedge \cdots \wedge a_n) \Rightarrow b$	$\equiv$	$\neg(a_1 \wedge \cdots \wedge a_n)$
notée	$\{a_1, \cdots, a_n\} \triangleright b$	notée	$\{a_1, \cdots, a_n\} \triangleright$
notée	$b \leftarrow a_1, \cdots, a_n$	notée	$\leftarrow a_1, \cdots, a_n$

Les clauses de Horn jouent un rôle particulier en programmation logique :

- les ***programmes logiques*** sont des ensembles de clauses de Horn de la forme $\Gamma \triangleright \Delta$ où Δ est un singleton :
$$(a_1 \wedge \cdots \wedge a_n) \Rightarrow b \qquad (n \geq 0)$$
que l'on note souvent :
$$b \leftarrow a_1, \cdots, a_n$$
- les ***requêtes*** sont des clauses négatives :
$$\neg a_1 \vee \cdots \vee \neg a_m \qquad (m \geq 1)$$
ou de manière équivalente :
$$\neg(a_1 \wedge \cdots \wedge a_m) \qquad (m \geq 1)$$
que l'on note souvent :
$$\leftarrow a_1, \cdots, a_m$$

On définit le type des clauses définies comme suit :

```
type 'a cl = { lit_pos : 'a ;
               lit_neg : 'a list};;
```

1.7.1 Sémantique par point fixe (Chaînage avant)

On introduit un opérateur de chaînage avant qui permet de caractériser, étant donné un ensemble I d'atomes, les atomes que l'on pourra déduire de I en appliquant les clauses qui apparaissent dans P :

$$\boxed{T_P(I) = \{a \mid \exists\ a \leftarrow b_1, \cdots, b_m \in P\ ,\ \{b_1, \cdots, b_m\} \subseteq I\}}$$

On définit alors la sémantique par point fixe d'un programme logique P en considérant l'ensemble $T_P^{\uparrow\omega}$ où :

$$\boxed{T_P^{\uparrow 0} = \emptyset \qquad T_P^{\uparrow n+1} = T_P(T_P^{\uparrow n}) \qquad T_P^{\uparrow\omega} = \bigcup_{n\geq 0} T_P^{\uparrow n}}$$

Considérons par exemple le programme suivant :

$$\begin{array}{l} p \leftarrow s \\ p \leftarrow q, r \\ r \leftarrow \\ q \leftarrow \\ t \leftarrow s \\ t \leftarrow w \\ u \leftarrow t \end{array} \tag{1}$$

$T_P^{\uparrow\omega}$ se calcule comme suit :

$$\begin{array}{l} T_P^{\uparrow 0} = \emptyset \\ T_P^{\uparrow 1} = T_P(T_P^{\uparrow 0}) = T_P(\emptyset) = \{r, q\} \\ T_P^{\uparrow 2} = T_P(T_P^{\uparrow 1}) = T_P(\{r, q\}) = \{r, q, p\} \\ T_P^{\uparrow 3} = T_P(T_P^{\uparrow 2}) = T_P(\{r, q, p\}) = \{r, q, p\} = T_P^{\uparrow 4} = \cdots = T_P^{\uparrow i} = \cdots \end{array}$$

Et on a donc :

$$T_P^{\uparrow\omega} = \bigcup_{n\geq 0} T_P^{\uparrow n} = \{r, q, p\}$$

En logique des propositions, on peut montrer qu'à partir d'un certain rang k (sur l'exemple, $k = 2$) on a :

$$T_P^{\uparrow k} = T_P^{\uparrow k+1} = T_P^{\uparrow k+2} = \cdots$$

et donc $T_P^{\uparrow\omega}$ peut s'obtenir en temps fini[3].

Exercice 1.12

[3] ce qui ne sera plus forcément le cas au premier ordre : par exemple, avec le programme $P = \{p(0) \leftarrow\ ;\ p(f(x)) \leftarrow p(x)\}$ on a $T_P^{\uparrow n} = \bigcup_{i\leq n} \{p(f^i(0))\}$ et donc $T_P^{\uparrow\omega}$ n'est pas fini.

1. Définir une fonction `in_L` *qui, étant données deux listes ℓ_1 et ℓ_2, détermine si tous les éléments de ℓ_1 sont dans ℓ_2.*

2. Définir une fonction `t` *qui, étant données une liste P de clauses définies (c'est à dire un programme logique) et une liste I d'atomes, calcule $T_P(I)$. On prendra soin de ne pas introduire de "doublons" dans la liste construite.*

3. Définir une fonction `forward` *qui, étant donnée une liste P de clauses définies (c'est à dire un programme logique), calcule $T_P^{\uparrow\omega}$. Pour cela, on pourra définir une fonction* `point_fixe` *qui étant donnés une fonction f et un élément x, applique f à x jusqu'à obtenir un point fixe. En d'autres termes, si $f(x) = x$ alors le résultat sera x, sinon on comparera $f(f(x))$ avec $f(x)$ ... et ainsi de suite. Le résultat de* `(point_fixe` *f x*`)` *est donc $f^n(x)$ où n est le plus petit entier tel que $f^n(x) = f(f^n(x)) = f^{n+1}(x)$.*

1.7.2 Sémantique déclarative

La sémantique déclarative de la programmation logique permet de caractériser la dénotation d'un programme sans faire intervenir la notion de "calcul effectif". Généralement, on exprime cette sémantique en terme de modèles : un ***modèle d'un programme*** P est une interprétation (en logique des propositions, une valuation) ν telle que chaque clause de P vérifie $\bar{\nu}(C) = \texttt{true}$. On peut montrer que tout programme logique P possède au moins un modèle – il s'agit de la ***base de Herbrand*** d'un programme logique P, notée B_P, qui correspond à l'ensemble des symboles propositionnels apparaissant dans P interprétés à `true` – et que l'intersection[4] de tous ses modèles est encore un modèle de P, appelé ***plus petit modèle de Herbrand*** et noté $\mathcal{M}_P$. De plus, on montre que les atomes qui appartiennent à $\mathcal{M}_P$ sont exactement les atomes qui sont conséquences sémantiques du programme P. On définit ainsi une sémantique déclarative d'un programme P en terme de plus petit modèle de P.

Par exemple le programme (1) admet les 6 modèles représentés ci-dessous :

$$\begin{array}{ccccc}
 & & \{u,t,w,s,q,r,p\} & & \\
 & / & & \backslash & \\
\{u,t,w,q,r,p\} & & & & \{u,t,s,q,r,p\} \\
 & \backslash & & / & \\
 & & \{u,t,q,r,p\} & & \\
 & & | & & \\
 & & \{u,q,r,p\} & & \\
 & & | & & \\
 & & \{q,r,p\} & &
\end{array}$$

On peut montrer que l'ensemble de ces modèles forme un treillis. Le plus petit modèle de Herbrand de ce programme est $\{q, r, p\}$.

Exercice 1.13

[4] Etant donnée une valuation ν sur un ensemble de variables propositionnelles E, on notera $\underline{\nu}$ le sous-ensemble de E contenant les variables propositionnelles p de E telles que $\bar{\nu}(p) = \texttt{true}$:

$$\underline{\nu} = \{p \in E \mid \bar{\nu}(p) = \texttt{true}\}$$

Réciproquement, à partir de $\underline{\nu}$ et E on peut reconstruire la valuation ν. On peut ainsi confondre une valuation et l'ensemble des variables propositionnelles qui sont vraies pour cette valuation et parler ainsi d'intersection de valuations et donc d'intersection de modèles.

1. *Définir une fonction* `make_conjonction` *qui, étant donnée une liste non vide de littéraux, construit la formule de type* `formul` *correspondant à la conjonction des littéraux apparaissant dans la liste.*

2. *Définir une fonction* `clause_to_formul` *qui, étant donnée une clause définie c, construit la formule de type* `formul` *correspondant à c.*

3. *Définir une fonction* `set_clause_to_formul` *qui, étant donnée une liste S de clauses définies, construit la formule de type* `formul` *correspondant à la conjonction des clauses de S.*

4. *Définir une fonction* `b` *qui, étant donnée une liste P de clauses définies, calcule la base de Herbrand B_P de P.*

5. *Définir une fonction* `intersection` *qui, étant données deux listes ℓ_1 et ℓ_2, construit la liste des éléments apparaissant dans ℓ_1 et dans ℓ_2.*

6. *Définir une fonction* `lists_intersection` *qui, étant donnée une liste ℓ de listes, construit la liste des éléments apparaissant dans chacune des listes de ℓ.*

7. *Définir une fonction* `m` *qui, étant donnée une liste P de clauses définies, construit la plus petit modèle de Herbrand $\mathcal{M}_P$ de P. On pourra réutiliser les fonctions définies pour calculer la forme normale disjonctive d'une formule en utilisant les tables de vérité ... notamment la fonction permettant de construire la liste de tous les modèles d'une formule.*

1.7.3 Sémantique opérationnelle (Chaînage arrière)

L'exécution d'un programme logique P à partir d'une requête $\leftarrow a_1, \cdots, a_m$ consiste à réfuter l'ensemble de clauses $P \cup \{\neg a_1 \vee \cdots \vee \neg a_m\}$ ce qui revient à prouver à partir de P la formule $a_1 \wedge \cdots \wedge a_m$.

La sémantique opérationnelle de la programmation logique décrit comment cette preuve est obtenue. On parle souvent de chaînage arrière pour exprimer le fait que l'on part de la formule dont on veut prouver la négation. Une requête $\leftarrow a'_1, \cdots, a'_m$ "réussit" si il existe une preuve de la clause vide par coupure à partir de $P \cup \{\neg a'_1 \vee \cdots \vee \neg a'_m\}$ obtenue en appliquant à chaque fois la règle de coupure à partir d'une clause définie de P et de la requête courante.

$$\left(\begin{array}{c} Cut \\ b = a'_i \end{array} \right) : \frac{b \leftarrow a_1, \cdots, a_n \qquad \leftarrow a'_1, \cdots, a'_i, \cdots, a'_m}{\leftarrow a'_1, \cdots, a'_{i-1}, a_1, \cdots, a_n, a'_{i+1}, \cdots, a'_m}$$

$$\left(\begin{array}{c} Cut \\ b \in \Gamma' \end{array} \right) : \frac{\Gamma \triangleright b \qquad \Gamma' \triangleright}{(\Gamma \cup \Gamma') \backslash \{b\}}$$

Si on prouve de cette manière la clause vide, alors cette preuve est une ***SLD-réfutation***. On définit ainsi la sémantique opérationnelle d'un programme logique P en terme d'ensemble S_P des succès de P :

$$\boxed{S_P = \{p \mid \exists \text{ une SLD-réfutation à partir de } \leftarrow p \text{ avec le programme } P\}}$$

Par exemple, si on considère le programme :

$$P = \{\triangleright r \ ; \ \triangleright q \ ; \{q, r\} \triangleright p \ ; \ \{s\} \triangleright p \ ; \ \{s\} \triangleright p_1 \ ; \ \{s\} \triangleright p_2\}$$

On a $p \in S_P$ puisque l'on peut construire la SLD-Réfutation de $P \cup \{p\triangleright\}$ suivante :

$$(Cut) : \cfrac{\triangleright q \quad (Cut) : \cfrac{\triangleright r \quad (Cut) : \cfrac{\{q,r\} \triangleright p \quad p\triangleright}{\{q,r\}\triangleright}}{\{q\}\triangleright}}{\emptyset \triangleright \emptyset}$$

Toutefois, ce mécanisme n'est pas encore tout à fait opérationnel puisqu'à chaque étape il faut choisir un atome dans la requête et une clause dans le programme[5]. Pour être tout à fait opérationnel il faut donc choisir une stratégie de sélection des atomes et des clauses. Nous allons choisir celle de Prolog qui consiste à considérer le premier atome de la requête et les clauses dans l'ordre de leur apparition dans le programme. Par exemple, à partir de la requête $\leftarrow a_1, \cdots, a_m$, si $a_1 \leftarrow b_1, \cdots, b_n$ est la première clause du programme P considéré (dont le littéral positif est a_1), alors la nouvelle requête à "résoudre" est : $\leftarrow b_1, \cdots, b_n, a_2, \cdots, a_m$. Si à l'issue de cette procédure on n'aboutit pas à une requête vide, il faudra alors reconsidérer, si c'est possible, une autre clause dans P. On construit ainsi un arbre de recherche. Par exemple, avec le programme (1), on a les deux arbres de recherche suivants :

$$\begin{array}{ccc}
 & p & \\
/ & & \backslash \\
s & & q, r \\
| & & | \\
\text{échec} & & r \\
 & & | \\
 & & \text{succès}
\end{array}
\qquad\qquad
\begin{array}{ccc}
 & u & \\
 & | & \\
 & t & \\
/ & & \backslash \\
s & & w \\
| & & | \\
\text{échec} & & \text{échec}
\end{array}$$

$$p \in S_P \qquad\qquad\qquad\qquad u \notin S_P$$

Les arbres de recherche ne sont pas forcément des arbres binaires (par exemple si plus de 2 clauses contiennent le même littéral positif) et on définit donc le type des arbres de recherche comme suit :

```
type 'a search_tree = Cons_search of ('a list) * ('a search_tree list);;
```

Exercice 1.14

1. *Définir une fonction* `sons` *qui, étant donné un arbre de recherche a, retourne la liste des arbres de recherche qui sont fils de la racine de a.*

2. *Définir une fonction* `make_search_tree` *qui, étant données une liste P de clauses définies et une liste R de littéraux positifs, construit une paire dont le premier élément est un booléen indiquant si l'arbre de recherche de R obtenu avec P contient une feuille de succès et donc le deuxième élément est l'arbre de recherche construit. Par exemple, les deux arbres de recherche présentés ci-dessus s'obtiendront comme suit :*

[5]Il s'agit de deux non-déterminismes de natures différentes. Le choix de l'atome est un non-déterminisme par indifférence ("*don't care*") et ne remet pas en cause l'obtention d'une "solution", tandis que le choix de la clause est un non-déterminisme par ignorance, plus délicat puisqu'il conditionne la forme de l'arbre et peut aboutir à la construction d'une branche infinie ("*don't know*").

```
# make_search_tree p ['p'];;
- : bool * char search_tree =
true,
Cons_search (['p'],
 [Cons_search (['s'], []);
  Cons_search (['q'; 'r'], [Cons_search (['r'], [Cons_search ([], [])])])])

# make_search_tree p ['u'];;
- : bool * char search_tree =
false,
Cons_search (['u'],
 [Cons_search (['t'], [Cons_search (['s'], []); Cons_search (['w'], [])])])
```

Par souci d'efficacité, on prendra soin d'éliminer les doublons dans les requêtes engendrées lors de la construction de l'arbre.

3. *Définir une fonction* `success` *qui étant donné un programme P calcule S_P. On pourra envisager les arbres de recherche de tous les éléments de B_P.*

4. *Vérifier que, lorsqu'elles terminent, les fonctions calculant $T_P^{\uparrow\omega}$, $\mathcal{M}_P$ et S_P (pour un programme P) calculent les mêmes ensembles.*

5. *Calculer $T_P^{\uparrow\omega}$, $\mathcal{M}_P$ et S_P pour le programme P suivant :*

$$P = \{a \leftarrow b, c \quad ; \quad b \leftarrow c, d \quad ; \quad b \leftarrow a, d \quad ; \quad c \leftarrow e \quad ; \quad d \leftarrow \quad ; \quad e \leftarrow\}$$

Que se passe-t-il si l'on permute les deux clauses $b \leftarrow c, d$ et $b \leftarrow a, d$?

2 Termes du premier ordre

La notion de termes du premier ordre permet de généraliser la plupart des objets manipulés en informatique. En effet, les termes servent à modéliser les structures de données (listes, piles, arbres, ...), les expressions à transformer (des programmes par exemple) ou encore les types ou les preuves. D'autre part, ils servent de base à la logique du premier ordre, utilisée pour exprimer des propriétés énoncées à l'aide de ces termes.

2.1 Définition des termes

La définition des termes se fait à partir d'une ***signature*** Σ, c'est à dire d'un ensemble de symboles qui pourront apparaître dans les termes, et d'un ensemble de variables V disjoint de Σ. Une signature Σ est un ensemble de symboles de fonction muni d'une application d'arité $ar : \Sigma \to \mathbb{N}$. Les termes sont définis de manière inductive à l'aide du système d'inférence du tableau 2. Les jugements sur lesquels portent les règles de ce système d'inférence sont des suites finies quelconques de symboles appartenant à $\Sigma \cup V$ (comme pour les formules de la logique des propositions, nous manipulons les termes au travers de leur arbre de syntaxe abstraite et nous utilisons donc ici des parenthèses pour préciser la forme de ces arbres). L'ensemble $T_\Sigma[V]$ des termes est donc l'ensemble des théorèmes du système d'inférence du tableau 2.

$$(\text{TV}) : \frac{}{x} \qquad\qquad (\text{TF}) : \frac{t_1 \cdots t_n}{f(t_1, \cdots, t_n)}$$

Table 2: Termes du premier ordre ($x \in V$, $f \in \Sigma$, $ar(f) = n$)

Etant donnés une signature Σ et un ensemble V de variables disjoint de Σ, l'ensemble $T_\Sigma[V]$ des termes est défini inductivement par :

- Un symbole de variable est un terme.
- Si $k \in \Sigma$ est un symbole d'arité nulle (c'est à dire une constante), alors k est un terme.
- Si $f \in \Sigma$ est un symbole d'arité $n > 0$, et si $t_1, \cdots, t_n$ sont des termes, alors $f(t_1, \cdots, t_n)$ est un terme.

On peut donc voir les symboles de fonction comme des constructeurs de termes.

Le type des termes peut être défini en OCAML 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 Σ. On remarquera ici qu'aucun contrôle n'est effectué sur le respect de l'arité des symboles de fonction. Ceci est du au fait qu'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.

Exercice 2.1

1. *L'ensemble des variables apparaissant dans un terme est défini comme suit :*

$$\vartheta(t) = \begin{cases} \{x\} & si\ t = x \\ \vartheta(t_1) \cup \cdots \cup \vartheta(t_n) & si\ t = f(t_1, \cdots, t_n) \end{cases}$$

   Définir en OCAML *la fonction* ϑ *de type* `('a,'b) term -> 'a list`.

2. *On dira qu'un terme* t *est* clos *s'il ne contient aucune variable (i.e., si* $\vartheta(t) = \emptyset$*). Définir en* OCAML *ua fonction de type* `('a,'b) term -> bool` *testant si un terme est clos.*

3. *On définit la* taille *d'un terme par une application* $\tau : T_\Sigma[V] \to \mathbb{N}$ *définie comme suit :*

$$\tau(t) = \begin{cases} 0 \ \ si\ \ t = x \\ 1 + \sum_{i=1}^{n} \tau(t_i) \ \ si\ \ t = f(t_1, \cdots, t_n) \end{cases}$$

   Définir en OCAML *ua fonction de type* `('a,'b) term -> int` *calculant la taille d'un terme.*

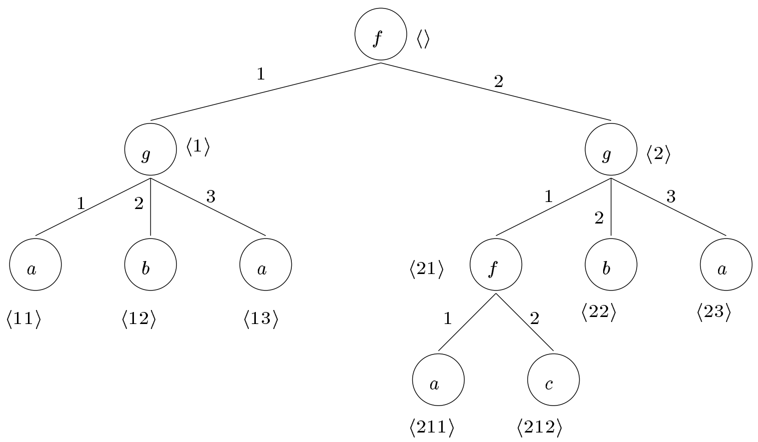

Figure 2: Arbre représentant le terme (2)

4. *On définit une relation d'ordre $\preceq_{ST}$ sur l'ensemble des termes correspondant à la notion de sous-terme. Un sous-terme d'un terme t est soit t lui-même, soit un sous-terme d'un terme t_i si t s'écrit $f(t_1, \cdots, t_n)$. Cette relation peut se définir à l'aide du système d'inférence ci-dessous :*

$$(ST_1) : \frac{}{t \preceq_{ST} t} \qquad (ST_2) : \frac{t \preceq_{ST} t_i}{t \preceq_{ST} f(t_1, \cdots, t_n)}$$

La relation $\preceq_{ST}$ peut se définir en OCAML *en remarquant que $t_1 \preceq_{ST} t_2$ si $t_1 = t_2$ ou si $t = f(t_1, \cdots, t_n)$ et s'il existe un terme t_i dont t soit un sous-terme. Définir une fonction* `subterm` *de type* `('a,'b) term -> ('a,'b) term -> bool` *qui implante la relation $\preceq_{ST}$.*

2.2 Termes et arbres

Nous allons voir qu'il est possible de donner une définition différente des termes qui repose sur la notion de domaine d'arbre qui va nous permettre de définir quelques opérations syntaxiques sur les termes.

Si l'on considère une signature Σ contenant les symboles a, b, c, f et g d'arités respectives 0, 0, 0, 2 et 3, alors :

$$f(g(a, b, a), g(f(a, c), b, a)) \tag{2}$$

est un terme puisqu'il existe un arbre de dérivation de ce terme avec les règles du tableau 2 :

$$(\mathrm{TF}) : \frac{(\mathrm{TF}) : \dfrac{(\mathrm{TF}) : \dfrac{}{a} \; (\mathrm{TF}) : \dfrac{}{b} \; (\mathrm{TF}) : \dfrac{}{a}}{g(a, b, a)} \quad (\mathrm{TF}) : \dfrac{(\mathrm{TF}) : \dfrac{(\mathrm{TF}) : \dfrac{}{a} \; (\mathrm{TF}) : \dfrac{}{b}}{f(a, c)} (\mathrm{TF}) : \dfrac{}{b} \; (\mathrm{TF}) : \dfrac{}{a}}{g(f(a, c), b, a)}}{f(g(a, b, a), g(f(a, c), b, a))}$$

Comme nous l'avons déjà dit cet arbre correspond en fait à l'arbre de syntaxe abstraite du terme (2) que l'on peut aussi représenter par l'arbre de la figure 2. Il est possible de numéroter les arcs issus de chaque noeud de cet arbre de gauche à droite à partir de 1. Cette numérotation des arcs permet d'associer à chaque noeud n un chemin constitué des numéros des arcs du chemin de la racine à n. La racine d'un arbre sera associée au chemin "vide". La figure 2 illustre cette représentation. L'ensemble des chemins associés aux noeuds de l'arbre représentant un terme t

$$(P_1) : \frac{}{\langle\rangle \preceq_{pref} c_1} \qquad (P_2) : \frac{c_1 \preceq_{pref} c_2}{\langle n\rangle.c_1 \preceq_{pref} \langle n\rangle.c_2}$$

Table 3: Relation $\preceq_{pref}$ ($n \in \mathbb{N}_+$, $c_1, c_2 \in \mathbb{N}_+^\star$)

est appelé le domaine d'arbre du terme t, noté $\mathcal{O}(t)$. Par exemple, le domaine d'arbre du terme (2) est :

$$\mathcal{O}(t) = \{\langle\rangle, \langle 1\rangle, \langle 11\rangle, \langle 12\rangle, \langle 13\rangle, \langle 2\rangle, \langle 21\rangle, \langle 211\rangle, \langle 212\rangle, \langle 22\rangle, \langle 23\rangle\}$$

Plus formellement, en notant $\mathbb{N}_+$ l'ensemble des entiers strictement positifs et $\mathbb{N}_+^\star$ l'ensemble des suites finies d'éléments de $\mathbb{N}_+$, un ***chemin*** est un élément de $\mathbb{N}_+^\star$.

En OCAML, on définit le type des chemins par :

```
type chemin = Empty_path | Cons_path of int*chemin;;
```

Exercice 2.2 *Définir deux fonctions* `length_path` *et* `append_path` *permettant respectivement de calculer la longueur d'un chemin et de concaténer deux chemins.*

Si c_1 et c_2 sont des éléments de $\mathbb{N}_+^\star$, on notera $c_1.c_2$ leur concaténation. Cette opération est associative et admet la suite vide pour élément neutre à gauche et à droite. Enfin, il est possible de définir une relation d'ordre permettant de caractériser la notion de préfixe :

$$\boxed{\forall c_1, c_2 \in \mathbb{N}_+^\star \quad c_1 \preceq_{pref} c_2 \quad \text{si et seulement si} \quad \exists c_3 \in \mathbb{N}_+^\star \quad c_1.c_3 = c_2}$$

On montre aisément que cette relation définit bien une relation d'ordre. Il s'agit seulement d'un ordre partiel puisque deux suites quelconques ne sont pas nécessairement comparables (c'est le cas par exemple pour les suites $\langle 23\rangle$ et $\langle 45\rangle$). Il est aussi possible de donner une définition équivalente de $\preceq_{pref}$ à partir d'un système d'inférence portant sur des jugements de la forme $c_1 \preceq_{pref} c_2$. On montre en effet que c_1 est un préfixe de c_2 si et seulement si le jugement $c_1 \preceq_{pref} c_2$ admet un arbre de dérivation à partir des règles du tableau 3.

Exercice 2.3

Définir une fonction `est_prefixe` *de type* `chemin -> chemin -> bool` *qui implante la relation* $\preceq_{pref}$.

On peut donc définir $\mathcal{O}$ comme une fonction de $T_\Sigma[V]$ dans $\wp(\mathbb{N}_+^\star)$ (où $\wp(E)$ désigne l'ensemble des parties de E) : Pour tout terme t, le ***domaine d'arbre*** de t est obtenu à partir de la fonction $\mathcal{O} : T_\Sigma[V] \to \wp(\mathbb{N}_+^\star)$ définie par :

$$\boxed{\mathcal{O}(t) = \begin{cases} \{\langle\rangle\} & \text{si } t = x \in V \\ \{\langle\rangle\} \cup \bigcup_{i=1}^{n} \{\langle i\rangle.c \mid c \in \mathcal{O}(t_i)\} & \text{si } t = f(t_1, \cdots, t_n) \end{cases}}$$

Exercice 2.4

1. *Définir une fonction* `dispatch` *de type* `int->chemin list->chemin list` *qui permet d'ajouter un entier e en tête des chemins éléments d'une liste de chemins ℓ.*

2. *Définir une fonction* `numbering_from` *de type :*

   ```
   int -> chemin list list -> chemin list list
   ```

   qui permet d'ajouter, étant donnés un entier n et une liste de listes de chemins :

   - *l'entier n en tête des chemins éléments de la 1-ère liste de listes de chemins ℓ*
   - *l'élément $n+1$ en tête des chemins éléments de la 2-ième liste de listes de chemins ℓ*
   - *...*
   - *l'élément $n+(k-1)$ en tête des chemins éléments de la k-ième liste de listes de chemins ℓ*

3. *En déduire une fonction* `ext_term`*, de type* `('a,'b) term -> chemin list`*, qui calcule le domaine d'arbre $\mathcal{O}(t)$ d'un terme t.*

A tout terme t, on peut associer une fonction A^t de $\mathcal{O}(t)$ dans $\Sigma \cup V$, c'est à dire une fonction partielle de $\mathbb{N}_+^\star$ dans $\Sigma \cup V$:

$$\boxed{A^t : \mathcal{O}(t) \subset \mathbb{N}_+^\star \to \Sigma \cup V}$$

telle que si $c \in \mathcal{O}(t)$, $A^t(c)$ désigne le symbole appartenant à $\Sigma \cup V$ apparaissant dans le noeud de l'arbre représentant le terme t auquel on accède par le chemin c.

$$\boxed{\begin{array}{ll} A^t(\langle\rangle) = x & \text{si } t = x \in V \\ A^t(\langle i\rangle.c) = A^{t_i}(c) & \text{si } t = f(t_1, \cdots, t_n) \quad (1 \le i \le n \text{ et } c \in \mathcal{O}(t_i)) \end{array}}$$

Par exemple, si l'on reconsidère le terme (2), on aura $A^t(\langle 21\rangle) = f$.

Exercice 2.5

L'implantation de la fonction A^t s'obtient en définissant tout d'abord un type somme correspondant à $\Sigma \cup V$:

```
type ('a,'b) sig_var = Fct of 'b | Var of 'a;;
```

Il suffit à présent de "parcourir" le terme t en suivant le chemin passé en argument. Une exception `Undefined_path` *sera levée si ce chemin n'appartient pas au domaine d'arbre du terme considéré. Définir une fonction* `occ_symb` *de type* `('a,'b) term -> chemin -> ('a,'b) sig_var` *qui implante A^t.*

Tout ensemble de chemins ne définit pas nécessairement un domaine d'arbre. Par exemple, $\{\langle 1\rangle, \langle 23\rangle\}$ n'est clairement pas un domaine d'arbre puisque d'une part aucun chemin ne correspond à la racine et d'autre part les chemins $\langle 2\rangle$, $\langle 21\rangle$ et $\langle 2,2\rangle$ ne figurent pas dans l'ensemble et donc le chemin $\langle 23\rangle$ ne peut pas correspondre à un chemin dans un arbre. Toutefois, par abus de notation, nous noterons $\mathcal{O}^{-1}(E)$ le terme t dont le domaine d'arbre est E : cette fonction n'est bien sûr définie que si E correspond bien au domaine d'arbre d'un terme. Un domaine d'arbre est un ensemble de chemins qui satisfait, par construction, certaines propriétés.

> Soit A un arbre. Le domaine d'arbre de A est un ensemble de chemins $E \subseteq \mathbb{N}_+^\star$ tel que pour tout chemin c :
>
> 1. pour tout chemin c' tel que $c' \preceq_{pref} c$, si $c \in E$, alors $c' \in E$
> 2. quels que soient les entiers i et j tels que $1 \le i \le j$, si $c.\langle j\rangle \in E$ alors $c.\langle i\rangle \in E$

On montre alors qu'étant donné un terme t, l'ensemble $\mathcal{O}(t)$ est un domaine d'arbre vérifiant une propriété supplémentaire relative au respect de l'arité des symboles de fonction.

Pour tout terme t, $\mathcal{O}(t)$ est un domaine d'arbre tel que si $c \in \mathcal{O}(t)$ et $ar(A^t(c)) = n$, alors les chemins $c.\langle 1\rangle$, $c.\langle 2\rangle$, ..., $c.\langle n\rangle$ appartiennent à $\mathcal{O}(t)$ et le chemin $c.\langle n+1\rangle$ n'appartient pas à $\mathcal{O}(t)$.

La notion de sous-terme d'un terme t, introduite dans l'exercice 2.1, peut être redéfinie à partir de $\mathcal{O}(t)$. Etant donné un ensemble de chemins E et un chemin c, on note $E \ominus c$ l'ensemble :

$$\boxed{E \ominus c = \{c' \in \mathbb{N}_+^\star \mid c.c' \in E\}}$$

$E \ominus c$ contient donc les chemins de E qui sont préfixés par c auxquels on a tronqué le préfixe c. On montre tout d'abord que cette transformation préserve les propriétés de domaine d'arbre d'un terme.

Si t est un terme, alors pour tout chemin $c \in \mathcal{O}(t)$, $\mathcal{O}(t) \ominus c$ est le domaine d'arbre d'un terme t' tel que $\forall c' \in \mathcal{O}(t) \ominus c \quad A^{t'}(c) = A^t(c.c')$

On montre alors la proposition suivante.

$$\boxed{\forall t \in T_\Sigma[V] \quad \forall c \in \mathcal{O}(t) \quad \mathcal{O}^{-1}(\mathcal{O}(t) \ominus c) \preceq_{ST} t}$$

Etant donné un chemin $c \in \mathcal{O}(t)$, on notera $t_{/c}$ le sous-terme de t dont le domaine d'arbre est $\mathcal{O}(t) \ominus c$:

$$\boxed{\forall t \in T_\Sigma[V] \quad \forall c \in \mathcal{O}(t) \quad \mathcal{O}^{-1}(\mathcal{O}(t) \ominus c) = t_{/c}}$$

Exercice 2.6

1. *Le domaine d'arbre du sous-terme $t_{/c}$ d'un terme t à l'occurrence c se calcule à l'aide d'une fonction* `sup_prefixe` *de type* `int->chemin->chemin` *permettant de supprimer les n premiers éléments d'un chemin. Définir cette fonction.*

2. *$\mathcal{O}(t_{/c})$ se calcule alors avec la fonction* `ext_subterm` *de type* `('a,'b) term->chemin->chemin list` *en construisant le domaine d'arbre $\mathcal{O}(t)$, puis en supprimant de cet ensemble tous les chemins qui ne sont pas préfixés par c et enfin en supprimant des ces chemins les n premiers éléments où n est la longueur du chemin c. Définir une telle fonction.*

La définition des termes exprimée en terme de domaines d'arbre permet d'introduire la notion de ***greffe***. Greffer un terme t' à l'occurrence c d'un terme t consiste à remplacer "dans" t le sous-terme $t_{/c}$ par t'. On notera $t[c \leftarrow t']$ le terme ainsi obtenu. Plus formellement cette opération est définie par :

$$\boxed{t[c \leftarrow t'] = \begin{cases} t' & \text{si } c = \langle\rangle \\ f(t_1, \cdots, t_{i-1}, t_i[c' \leftarrow t'], t_{i+1}, \cdots, t_n) & \text{si } t = f(t_1, \cdots, t_n) \\ & \text{et } c = \langle i.c'\rangle \ (1 \leq i \leq n) \end{cases}}$$

Par exemple, si t est le terme (2), on aura :

$$t[\langle 21\rangle \leftarrow g(a,b,c)] = f(g(a,b,c), g(g(a,b,c),b,a))$$

Cette opération de greffe peut se faire à partir du domaine d'arbre d'un terme puisque l'on montre que :

Si t et t' sont des termes et si $c \in \mathcal{O}(t)$ alors :

$$\mathcal{O}(t[c \leftarrow t'])) = (\mathcal{O}(t) \backslash \{c' \in \mathbb{N}_+^\star \mid c \preceq_{pref} c'\}) \cup \{c.c' \mid c' \in \mathcal{O}(t')\} \tag{3}$$

Exercice 2.7

1. *Définir une fonction* `greffer` *de type* `('a,'b) term->chemin->('a,'b) term->('a,'b) term` *qui implante la greffe d'un terme.*

2. *En utilisant la proposition (3), définir une fonction de type* `('a,'b)term->chemin->('c,'d) term->chemin list` *qui permet de calculer le domaine d'un terme greffé. En observant le type de cette fonction, on remarquera que lorsque l'on manipule directement les chemins de deux termes, rien n'impose que ces deux termes soient construits à partir de la même signature.*

2.3 Substitutions

Une ***substitution*** est une opération sur les termes permettant de remplacer certaines variables d'un terme par un terme. Plus formellement, l'ensemble Θ des substitutions est l'ensemble des fonctions de V dans $T_\Sigma[V]$:

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

On notera s_{id} la substitution identité, c'est à dire la substitution telle que pour toute variable $x \in V$, $s_{id}(x) = x$. Une substitution θ se prolonge de façon unique en une fonction $\hat{\theta}$ de $T_\Sigma[V]$ dans $T_\Sigma[V]$ définie par :

$$\begin{array}{l}
\forall x \in V \quad \hat{\theta}(x) = \theta(x) \\
\forall f \in \Sigma \ \forall t_1, t_2, \cdots, t_{ar(f)} \in T_\Sigma[V] \\
\qquad \hat{\theta}(f(t_1, t_2, \cdots, t_{ar(f)})) = f(\hat{\theta}(t_1), \hat{\theta}(t_2), \cdots, \hat{\theta}(t_{ar(f)}))
\end{array}$$

$$\begin{tikzcd}
V \arrow[r, "\theta"] \arrow[d] & T_\Sigma[V] \\
T_\Sigma[V] \arrow[ur, dashed, "\hat{\theta}"'] &
\end{tikzcd}$$

Appliquer une substitution à un terme c'est donc instancier certaines de ses variables par des termes et donc faire augmenter sa taille :

$$\forall \theta \in \Theta \quad \forall t \in T_\Sigma[V] \quad \tau(\hat{\theta}(t)) \geq \tau(t)$$

La plupart du temps, les substitutions que nous manipulons correspondent en fait 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 de couples (x, t) où x est la variable à remplacer par le terme t. Nous noterons donc :

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

la substitution θ de domaine $dom(\theta) = \{x_1, \cdots, x_n\}$ telle que $\theta(x_i) = t_i$ (pour $1 \leq i \leq n$) et telle que $\theta(y) = y$ pour tout $y \notin dom(\theta)$. Appliquer la substitution θ à un terme t, c'est donc remplacer chaque occurrence des variables x_i par les termes t_i.

La représentation des substitutions par des listes de couples de la forme (x, t) n'est qu'un moyen de donner une représentation extensionnelle d'une fonction. Cette “notation” ne doit pas nous faire oublier qu'une substitution θ est avant tout une fonction et est donc déterministe : pour toute variable x, $\theta(x)$ existe et est unique. Aussi, une liste de couples $[(x_1, t_1), \cdots, (x_n, t_n)]$ ne représente une substitution de domaine fini que si les variables x_1, ..., x_n sont deux à deux distinctes.

Exercice 2.8

On représente les subtitutions par des listes d'association.

1. *Définir une fonction* `apply_subst_V` *de type :*

```
('a * ('b, 'a) term) list -> 'a -> ('b, 'a) term
```

qui permet d'appliquer une substitution à une variable.

2. *Définir une fonction* `apply_subst_T` *de type :*

```
('a * ('b, 'a) term) list -> ('b, 'a) term -> ('b, 'a) term
```

qui permet d'appliquer une substitution à un terme.

La ***composition*** des deux substitutions représentées par les listes :

$$\sigma_1 = [(x_1, t_1), \cdots, (x_n, t_n)] \qquad \sigma_2 = [(v_1, t'_1), \cdots, (v_k, t'_k)]$$

est la substitution représentée par la liste :

$$\sigma_2\sigma_1 = [(x_1, \widehat{\sigma_2}(t_1)), \cdots, (x_n, \widehat{\sigma_2}(t_n)), (v_1, t'_1), \cdots, (v_k, t'_k)]$$

dans laquelle on a supprimé toutes les couples $(x_i, \widehat{\sigma_2}(t_i))$ tels que $x_i = \widehat{\sigma_2}(t_i)$ (puisque dans ce cas x_i n'appartient plus au domaine de $\sigma_2\sigma_1$) et tous les couples (v_j, t'_j) tels que $v_j \in \{x_1, \cdots, x_n\}$ (il est en effet inutile de conserver ces couples puisque la substitution σ_1 est appliquée avant la substitution σ_2).

Exercice 2.9

Définir une fonction `comp_subst` *de type :*

```
('a*('b,'a) term) list -> ('a*('b,'a) term) list -> ('a*('b,'a) term) list
```

qui permet d'obtenir la composition $\sigma_2\sigma_1$ des deux substitutions σ_1 et σ_2.

Etant donnée une substitution σ, il est parfois utile de pouvoir “enrichir” σ de telle manière qu'une nouvelle variable soit affectée par la substitution ainsi enrichie. C'est notamment le cas lorsque l'on construit incrémentalement une substitution. On note cette opération $\sigma \oplus (x, t)$ où t est le terme tel que $(\sigma \oplus (x, t))(x) = t$. Cette opération n'est correcte que si ce couple existe déjà dans σ, c'est à dire si $\sigma(x) = t$, et dans ce cas σ reste inchangée, c'est à dire $\sigma \oplus (x, t) = \sigma$, ou bien si $\sigma(x) = x$ c'est à dire si x n'appartient pas au domaine de σ. En d'autres termes, si $\sigma(x) = t'$ avec $t' \neq x$ et $t \neq t'$, l'opération $\oplus$ n'est pas définie (ou échoue).

Exercice 2.10

Définir une fonction `add_subst` *de type :*

```
('a * 'b) list -> 'a * 'b -> ('a * 'b) list
```

qui implémente l'opération $\oplus$.

2.4 Filtrage – Unification

On peut définir un préordre[6] (relation réflexive et transitive) sur $T_\Sigma[V]$ comme suit :

$$\boxed{t_2 \preceq t_1 \text{ si et seulement si il existe une substitution } \sigma \text{ telle que } \hat{\sigma}(t_2) = t_1}$$

et lorsque $t_2 \preceq t_1$, on dit que le terme t_1 est filtré par le terme t_2. L'opération de ***filtrage*** consiste donc à construire, si elle existe, une telle substitution. Le principe du filtrage de t_1 par t_2 est simple. Il suffit d'appeler une fonction auxiliaire avec la substitution identité (i.e., la liste vide) et les deux termes t_1 et t_2. Etant donnés une substitution σ et deux termes t_1 et t_2, cette fonction procède comme suit :

- si t_2 est une variable x, alors le résultat est la substitution $\sigma \oplus (x, t_1)$
- si $t_2 = f(t'_1, \cdots, t'_n)$, alors :
  - si $t_1 = f(t''_1, \cdots, t''_n)$, alors :
    - il suffit de procéder à un appel récursif avec σ, t'_1 et t''_1 comme argument pour obtenir la substitution σ_1
    - puis de procéder à nouveau à un appel récursif avec σ_1, t'_2 et t''_2 comme argument pour obtenir la substitution σ_2
    - et ainsi de suite ...
  - sinon t_1 n'est pas filtré par t_2

La substitution résultat est donc construite incrémentalement à l'aide de l'opération $\oplus$. Le filtrage peut échouer, soit lorsque l'on a détecté un conflit entre symboles de fonction, soit lorsque la fonction $\oplus$ renvoie une erreur.

Exercice 2.11
Définir une fonction `match_term` *de type :*

```
('a, 'b) term -> ('a, 'c) term -> ('b * ('a, 'c) term) list
```

qui réalise le filtrage de deux termes.

L'unification est une opération syntaxique sur les termes, utilisée à la fois pour prouver et pour calculer. L'unification est une propriété définie sur les termes : deux termes t_1 et t_2 sont ***unifiables*** s'il existe une substitution θ telle que $\widehat{\theta}(t_1) = \widehat{\theta}(t_2)$.

Décider si deux termes t_1 et t_2 sont unifiables revient à résoudre l'équation $t_1 = t_2$. Plus généralement, l'algorithme d'unification permet de résoudre un système d'équations $\mathcal{S}$ entre termes en transformant $\mathcal{S}$ en un système de la forme $\cup_1^n\{x_i = t_i\}$, tel que les variables x_i soient deux à deux distinctes et tel que $\{x_1, \cdots, x_n\} \cap \cup_1^n \vartheta(t_i) = \emptyset$, si $\mathcal{S}$ admet une solution (et dans ce cas, ce système correspond à une substitution) ou en un système noté $\{\bot\}$ si $\mathcal{S}$ n'admet pas de solution. Un tel système est dit résolu. L'algorithme d'unification consiste en la définition d'une relation de transition, notée $\rightsquigarrow$, et résoudre un système d'équations $\mathcal{S}$ entre termes revient alors à construire une suite de transitions $\mathcal{S} \rightsquigarrow \mathcal{S}_1 \rightsquigarrow \cdots \rightsquigarrow \mathcal{S}_n$ telle que le système $\mathcal{S}_n$ soit résolu. Cet algorithme est classique et est présenté dans le tableau 4.

[6]Par contre, $\preceq$ ne définit pas un ordre, puisque $t_1 \preceq t_2$ et $t_2 \preceq t_1$ n'implique pas forcément $t_1 = t_2$. Par exemple, si on considère les deux termes $g(x)$ et $g(y)$, on a d'une part, $g(x) \preceq g(y)$ puisque la substitution θ_1 qui associe y à x et laisse inchangées les autres variables vérifie bien $\hat{\theta}_1(g(x)) = g(y)$ et d'autre part, on a $g(y) \preceq g(x)$ puisque la substitution θ_2 qui associe x à y et laisse inchangées les autres variables vérifie bien $\hat{\theta}_2(g(y)) = g(x)$. Par contre ces deux termes sont "équivalents à un renommage près des variables".

$\{f(t_1, \cdots, t_n) = f(t'_1, \cdots, t'_n)\} \cup E$	$\overset{\text{décomposition}}{\rightsquigarrow}$	$\{t_1 = t'_1, \cdots, t_n = t'_n\} \cup E$
$\{f(t_1, \cdots, t_n) = g(t'_1, \cdots, t'_n)\} \cup E$	$\overset{\text{conflit}}{\rightsquigarrow}$	$\{\perp\}$
$\{x = t\} \cup E$	$\overset{\text{élimination}}{\rightsquigarrow}$	$\{x = t\} \cup E[x \leftarrow t]$ si $x \in \vartheta(E)$ et $x \notin \vartheta(t)$
$\{x = t\} \cup E$	$\overset{\text{occurrence}}{\rightsquigarrow}$	$\{\perp\}$ si $x \in \vartheta(t)$
$\{t = t\} \cup E$	$\overset{\text{effacement}}{\rightsquigarrow}$	E
$\{t = x\} \cup E$	$\overset{\text{inversion}}{\rightsquigarrow}$	$\{x = t\} \cup E$ si $t \notin V$

Table 4: Algorithme d'unification

Toutefois, sous cette forme, il n'est pas directement opérationnel si l'on ne précise pas une stratégie d'application des règles. Aussi, l'algorithme que nous allons utiliser se base sur la structure des deux termes t_1 et t_2 à unifier :

- si t_1 est la variable x, alors :
  - si x apparaît dans le terme t_2 et si $t_2 \neq x$, alors l'unification échoue[7] – si $t_2 = x$ alors l'unification réussit avec la substitution identité
  - sinon, l'unification réussit avec la substitution $[(x, t_2)]$
- si $t_1 = f(t'_1, \cdots, t'_n)$, alors :
  - si t_2 est une variable, alors on procède de manière similaire au cas précédent
  - sinon si $t_2 = f(t''_1, \cdots, t''_n)$ alors il faut "unifier" l'ensemble de paires de termes $\{(t'_1, t''_1), \cdots, (t'_n, t''_n)\}$ comme suit :
    - on essaie d'unifier les deux termes t'_1 et t''_1 pour obtenir la substitution σ_1
    - puis, si cette unification réussit, on essaie d'unifier les deux termes $\widehat{\sigma_1}(t'_2)$ et $\widehat{\sigma_1}(t''_2)$ pour obtenir la substitution σ_2
    - puis, si cette unification réussit, on essaie d'unifier les deux termes $\widehat{\sigma_2\sigma_1}(t'_2)$ et $\widehat{\sigma_2\sigma_1}(t''_2)$ pour obtenir la substitution σ_3
    - et ainsi de suite ... si toutes ces étapes d'unification réussissent alors la substitution résultat est $\sigma_n \cdots \sigma_2\sigma_1$
  - sinon ($t_2 = g(t''_1, \cdots, t''_k)$ avec $f \neq g$) l'unification échoue

Aussi, lorsque l'unification réussit, la substitution résultat est construite incrémentalement par composition des substitutions construites à chaque étape.

Exercice 2.12

Définir une fonction `unify` *de type :*

[7] En effet, considérons par exemple les deux termes $t_1 = f(x)$ et $t_2 = x$. Unifier t_1 et t_2 revient à résoudre l'équation $x = f(x)$ qui n'admet pas de solution finie puisque la solution de cette équation est $x = f^\omega = f(f(f(f(f \cdots$ (qui est un terme rationnel infini). Or nous manipulons ici uniquement des termes finis.

```
('a, 'b) term * ('a, 'b) term -> ('b * ('a, 'b) term) list
```

qui réalise l'unification de deux termes.

2.5 Interprétation des termes

Nous allons à présent interpréter les termes. Tout d'abord les symboles de fonction vont être interprétés par des fonctions sur un certain domaine. Etant donnée une signature Σ, une Σ-algèbre $\mathcal{A}$ de domaine A est une application qui à tout symbole de fonction de Σ d'arité n associe une application $f^{\mathcal{A}}$ de A^n dans A. Pour pouvoir interpréter les termes il suffit à présent de connaître les valeurs associées aux variables : une *valuation* v sur un domaine A est une fonction de V dans A. On note $\mathcal{V}[A]$ l'ensemble de ces valuations :

$$\mathcal{V}[A] = \{v : V \to A\}$$

Le schéma d'interprétation des termes dans une Σ-algèbre $\mathcal{A}$ de domaine A est alors défini comme suit. Etant donnée une valuation v dans A, un terme t est interprété en une valeur $[t]_v \in A$ définie par :

- si t est une variable x, alors $[x]_v = v(x)$
- si $t = f(t_1, \cdots, t_n)$, alors $[t]_v = f^{\mathcal{A}}([t_1]_v, \cdots, [t_n]_v)$.

On remarquera que l'ensemble des substitutions Θ n'est rien d'autre que l'ensemble $\mathcal{V}[T_\Sigma[V]]$ des valuations dont le domaine est l'ensemble des termes et que l'application d'une substitution définit donc une interprétation des termes par des termes.

Exercice 2.13

Définir une fonction `interp_term` *de type :*

```
('a->'b) -> ('c->'b list->'b) -> ('a,'c) term -> 'b
```

qui, étant donnée une valuation $v : a \to b$ (a correspond au type des variables et b au domaine de l'interprétation), une Σ-algèbre $\mathcal{A}$ de domaine b qui a tout symbole de fonction d'arité n (c représente le type de la signature Σ) associe une fonction de b^n (représenté par une liste d'élements de type b) dans b et un terme t, calcule le résultat (de type b) de l'interprétation de t.

Exercice 2.14

Les éléments de l'ensemble $\mathcal{F}_p$ des formules de la logique des propositions sont en fait des termes construits à partir d'un ensemble Σ_p de symboles propositionnels jouant le rôle des variables (ces symboles sont d'ailleurs parfois appelés des variables propositionnelles) et d'un ensemble $\Sigma_c = \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ de connecteurs logiques associés à une arité[8]. On a donc $\mathcal{F}_p = T_{\Sigma_c}[\Sigma_p]$ puisque le système d'inférence du tableau 2 contient les règles du système d'inférence du tableau 1 lorsque $V = \Sigma_p$ et $\Sigma = \Sigma_c$.

Il est possible d'interpréter les termes de cet ensemble dans la Σ_c-algèbre $\mathcal{A}_{\mathbb{B}}$ de domaine $\mathbb{B}$ telle que :

$$\neg^{\mathcal{A}_{\mathbb{B}}} = [\neg] \quad \wedge^{\mathcal{A}_{\mathbb{B}}} = [\wedge] \quad \vee^{\mathcal{A}_{\mathbb{B}}} = [\vee] \quad \Rightarrow^{\mathcal{A}_{\mathbb{B}}} = [\Rightarrow] \quad \Leftrightarrow^{\mathcal{A}_{\mathbb{B}}} = [\Leftrightarrow]$$

où $[\neg]$, $[\wedge]$, $[\vee]$, $[\Rightarrow]$ et $[\Leftrightarrow]$ sont les fonctions booléennes définies page 5. Dans ce contexte, une valuation ν est une fonction de Σ_p dans $\mathbb{B}$ (on retrouve exactement la définition des valuations

[8] 1 pour le symbole de négation $\neg$ et 2 pour les autres connecteurs de Σ_c

en logique des propositions) et le schéma d'interprétation des termes de $T_{\Sigma_c}[\Sigma_p]$ dans la Σ_c-algèbre $\mathcal{A}_{\mathbb{B}}$ correspond alors exactement au schéma d'interprétation des formules de la logique des propositions.

Dans ce qui suit, on considère donc une signature contenant uniquement des symboles de fonction d'arité inférieure ou égale à 2. Les variables et les symboles de fonction sont représentés par des chaînes de caractères.

1. *Définir un type somme* `terme`, *muni des constructeurs* `V`, `C`, `U` *et* `B` *correspondant respectivement aux variables, aux constantes, aux termes obtenus à partir d'un symbole de fonction unaire et aux termes obtenus à partir d'un symbole de fonction binaire.*

2. *Définir un type enregistrement* `'a algebre` *qui implémente les algèbres de domaine* `'a` *(les trois champs de cet enregistrement correspondent à l'interprétation des constantes, des symboles de fonction unaire et des symboles de fonction binaire).*

Une façon d'implémenter les valuations dans A, consiste à considérer une valuation comme une liste de liaisons définies par des paires de type `(string * 'a)` *qui permettent d'associer une valeur de type* `'a` *à une variable (c'est à dire un objet de type* `string`*).*

3. *Définir une fonction* `evalue` *d'évaluation des termes. Cette fonction a pour type :*

```
'a algebre -> terme -> (string * 'a) list -> 'a
```

4. *Définir* `algebre_prop` *comme étant l'algèbre $\mathcal{A}_{\mathbb{B}}$.*

5. *On souhaite définir une fonction* `est_tautologie` *qui étant donnés un terme t et la liste ℓ des propositions atomiques apparaissant dans t retourne* `true` *si t est une tautologie (c'est à dire si pour toute valuation, t s'évalue à* `true`*). Cette fonction utilisera donc la fonction* `evalue` *avec pour argument l'algèbre* `algebre_prop`*. Elle tentera de construire progressivement une valuation pour laquelle le terme t s'évalue à* `false`*. Pour cela, la fonction* `est_tautologie` *définira dans son corps une fonction* `test` *de type :*

```
(string * bool) list -> string list -> unit
```

dont le premier argument correspond à la valuation en cours de construction et dont le deuxième argument correspond à la liste des chaînes de caractères correspondant aux variables de t qui n'apparaissent pas dans le premier argument. Cette fonction lèvera une exception à définir si elle permet de construire une valuation (en associant une valeur à son deuxième argument) pour laquelle t s'interprète à `false` *et rendra* `()` *sinon. L'exception levée par la fonction* `test` *devra être rattrapée par la fonction* `est_tautologie`.

3 Logique de premier ordre

3.1 Formules de la logique du premier ordre

L'ensemble $\mathcal{F}_L$ des formules de la logique du premier ordre est défini inductivement, à partir des termes et d'une ensemble $\mathbb{P}$ de prédicats muni d'une fonction d'arité, comme suit :

- si t_1, ..., t_n sont des termes et si p est un symbole de $\mathbb{P}$ d'arité n, alors $p(t_1, \cdots, t_n) \in \mathcal{F}_L$ (cette formule est un atome).

- si $\varphi \in \mathcal{F}_L$ alors $\neg\varphi \in \mathcal{F}_L$

- si $\varphi, \psi \in \mathcal{F}_L$ alors $\varphi \wedge \psi$, $\varphi \vee \psi$ et $\varphi \Rightarrow \psi$ sont des éléments de $\mathcal{F}_L$.
- si si x est une variable et si $\varphi \in \mathcal{F}_L$ alors $\forall x\varphi \in \mathcal{F}_L$ et $\exists x\varphi \in \mathcal{F}_L$.

Le type des formules logiques du premier ordre peut se définir à partir des termes en utilisant les constructeurs suivants :

```
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` désigne le type des variables, `'b` le type des symboles de fonction, et `'c` le type des symboles de prédicats. Ici encore, on remarquera qu'aucun contrôle n'est effectué sur l'arité des symboles de prédicat.

Les quantificateurs qui peuvent apparaître dans les formules permettent de lier les variables ($\forall$ et $\exists$ sont des *binders*) et il est utile de pouvoir déterminer si une variable x est dans la portée d'un quantificateur ou si elle est *libre*. On définit l'ensemble $\vartheta_F(\varphi)$ des variables libres d'une formule $\varphi \in \mathcal{F}_L$ comme suit :

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

Savoir si l'occurrence d'une variable dans une formule est libre ou liée est nécessaire si l'on veut pouvoir appliquer une substitution à une formule. En effet, substituer une variable x par une terme t dans une formule φ pour obtenir une formule notée $\varphi[x := t]$ consiste à remplacer toutes les occurrences libres de x dans φ par le terme t. Toutefois, cette opération n'est correcte que si les variables apparaissant dans t ne sont pas liées dans φ. En effet, considérons par exemple la formule $\forall xp(x, y)$ dans laquelle nous souhaitons substituer la variable y par le terme $f(x)$. Un simple remplacement de y par $f(x)$ conduit à la formule $\forall xp(x, f(x))$ qui n'est pas correcte puisque l'occurrence de x dans le terme $f(x)$ devient liée du fait de la quantification $\forall x$. Il faut donc, avant d'effectuer la substitution, renommer les variables liées de φ qui apparaissent dans t. Sur l'exemple ci-dessus, ce renommage conduit à la formule $\forall zp(z, y)$ (qui est logiquement équivalente à la formule $\forall xp(x, y)$) sur laquelle on peut à présent effectuer la substitution de y par $f(x)$ de manière correcte et obtenir la formule $\forall zp(z, f(x))$ (qui n'est pas équivalente à la formule $\forall xp(x, f(x))$).

Exercice 3.1

Définir une fonction `single_subst_f` *de type :*

```
'a -> ('a, 'b) term -> ('a, 'b, 'c) form -> ('a, 'b, 'c) form
```

qui permet de calculer $\varphi[x := t]$. On supposera que les variables de t ne sont pas liées dans φ.

3.2 Interprétation des formules

Puisque, étant données une valuation et une algèbre de domaine A, on sait interpréter les termes en une valeur dans A, il suffit maintenant de disposer d'une interprétation des symboles de prédicat pour pouvoir interpréter les formules de $\mathcal{F}_L$ en une valeur dans $\mathbb{B}$. On enrichit donc la notion de $(\Sigma \cup \mathbb{P})$-algèbre qui est définie par :

- un domaine d'interprétation A
- une interprétation des symboles de fonction qui permet d'associer à tout $f \in \Sigma$ d'arité n une fonction $f^{\mathcal{A}} : A^n \to A$
- une interprétation des symboles de prédicat qui permet d'associer à tout $p \in \mathbb{P}$ d'arité n une fonction $p^{\mathcal{A}} : A^n \to \mathbb{B}$

Le schéma d'interprétation des formules est à présent défini comme suit :

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

Exercice 3.2

En utilisant la fonction `interp_term` *définie dans l'exercice 2.13, définir une fonction* `interp_form` *qui permet d'interpréter une formule de $\mathcal{F}_L$. Le type de cette fonction sera :*

```
    ('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 A, on a $[\varphi[x := a]]_v = \mathsf{true}$ 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 A tel que $[\varphi[x := a]]_v = \mathsf{true}$. 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.*

3.3 Programmation du noyau du logiciel Tarski's world

Le logiciel Tarski's world permet de construire un modèle et d'interpréter des formules logiques du premier ordre dans ce modèle.

3.3.1 Syntaxe

Les formules que le logiciel Tarski's world permet de vérifier sont des formules de la logique du premier ordre construites à partir :

- d'un ensemble fini $V = \{\texttt{U,V,W,X,Y,Z}\}$ de variables
- d'un ensemble fini $\Sigma = \{\texttt{A,B,C,D,E,F}\}$ de symboles de fonction correspondant à des constantes (i.e., dont l'arité est nulle)
- d'un ensemble fini :

$$\mathbb{P} = \left\{ \begin{array}{l} \texttt{Tet, Cub, Dodec, Small, Medium, Large, Smaller, Larger,} \\ \texttt{LeftOf, RightOf, BackOf, FrontOf, Between} \end{array} \right\}$$

de symboles de prédicat (se reporter à la documentation du logiciel pour l'arité de ces symboles)

En OCAML, on représente ces ensembles par les types suivants :

```
type variables = U | V | W | X | Y | Z;;

type cstes = A | B | C | D | E | F;;

type predicats =   Tet | Cub | Dodec | Small | Medium | Large
                 | Smaller | Larger | LeftOf | RightOf | BackOf
                 | FrontOf | Between;;
```

3.3.2 Interprétation des formules

Domaine d'interprétation Le logiciel Tarski's world permet de construire un modèle à partir d'une grille sur laquelle sont disposés des objets. Le domaine d'interprétation des modèles que le logiciel Tarski's world permet de construire est donc un ensemble d'objets placés sur une grille.

La grille est représentée par un tableau à deux dimensions dont les éléments sont des cases. Le type `case` des cases permet de distinguer les cases vides des cases contenant un objet.

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

Exercice 3.3

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.*

Outre sa position sur la grille, chaque objet – de type `objet` – est caractérisé par une forme (pyramide, cube, dodécaèdre) – type `figure` – et une taille (petit, moyen, grand) – type `taille`.

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

Interprétation des symboles de fonction Les seuls symboles de fonction considérés sont des constantes. Ces constantes servent à désigner des objets sur la grille. Comme nous l'avons vu, le domaine d'interprétation des symboles de constante est donc `int * int` (les coordonnées de l'objet sur la grille) lorsque la constante considérée a été introduite ou `Undef` sinon :

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

Exercice 3.4

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.

1. *Etant donnée une fonction* `i_f` *d'interprétation des symboles de constante, pour associer un objet à une 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 -> figure -> taille
   -> ('a -> interp_cste) -> 'a -> interp_cste
   ```

   qui permet d'effectuer cette opération.

2. *Définir une fonction* `sup_cste` *de type :*

   ```
        'a case array array -> 'b -> int -> int
   -> ('b -> interp_cste) -> 'b -> interp_cste
   ```

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

Interprétation des symboles de prédicat Nous ne redonnons pas ici la sémantique des prédicats que l'on peut utiliser pour construire des formules avec le logiciel Tarski's world (se reporter à la documentation du logiciel). Pour implanter cette sémantique en OCAML, il suffit de définir une fonction d'interprétation pour chaque prédicat. Lorsque l'arité des symboles de prédicat n'est pas respectée, une exception sera levée. Par exemple, la fonction d'interprétation du prédicat `Tet` peut être implantée comme suit :

```
let interp_Cub w l = match l with
   [x] -> (w.((fst x)).((snd x))).fig = Cube
 | _ -> raise Arity_error;;
```

Exercice 3.5

Définir toutes les fonctions d'interprétation des prédicats invoquées dans la fonction générale suivante :

```
let interp_pred w p = match p with
   Tet     -> (function l -> interp_Tet w l)
  | Cub     -> (function l -> interp_Cub w l)
  | Dodec   -> (function l -> interp_Dodec w l)
  | Small   -> (function l -> interp_Small w l)
  | Medium  -> (function l -> interp_Medium w l)
  | Large   -> (function l -> interp_Large w l)
  | Smaller -> (function l -> interp_Smaller w l)
  | Larger  -> (function l -> interp_Larger w l)
  | LeftOf  -> (function l -> interp_LeftOf w l)
  | RightOf -> (function l -> interp_RightOf w l)
  | BackOf  -> (function l -> interp_BackOf w l)
  | FrontOf -> (function l -> interp_FrontOf w l)
  | Between -> (function l -> interp_Between w l);;
```

3.3.3 Vérification d'une formule

Une fois la syntaxe et la sémantique du langage définies, il devient possible de vérifier une formule étant donnée une grille. Il suffit pour cela d'utiliser la fonction `interp_form` définie dans l'exercice 3.2. Toutefois cette fonction nécessite en argument deux fonctions permettant de parcourir le domaine d'interprétation considéré. Puisque, dans le cadre de l'interprétation que nous considérons ici, ce domaine est fini, ces deux fonctions se programment facilement.

Exercice 3.6

1. *Définir deux fonctions* `check_forall` *et* `check_exists` *ayant pour type :*

   ```
   'a ->
   ('a -> 'b) ->
   (cstes -> 'b list -> 'b) ->
   ('c -> 'b list -> bool) -> ('a, cstes, 'c) form -> bool
   ```

   La première 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 A, on a $[\varphi[x := a]]_v = \mathsf{true}$ et la deuxième est une fonction similaire qui permet d'indiquer si il existe un élément a appartenant au domaine d'interprétation A tel que $[\varphi[x := a]]_v = \mathsf{true}$.

2. *En déduire une fonction* `interp_tarski` *de type :*

   ```
       objet array array
   -> (cstes -> (int * int) list -> int * int)
   -> ('a, cstes, predicats) form
   -> bool
   ```

   qui permet de vérifier qu'une grille est un modèle d'une formule.