Concours d'admission de l'École polytechnique et des ENS (X-ENS), France

Session 2024, filière MPI

Informatique C

Durée : 4 heures

Ce que traite le sujet

- Partie I : équivalences de littéraux
- Partie II : équivalences avec une conjonction de deux littéraux
- Partie III : égalités et différences entre entiers
- Partie IV : égalités et différences d'expressions

Sujet reproduit sans retouche à partir de la page suivante.

Résumé des parties, chapitres de cours liés et autres années :
prepahelp.com/annales/x-ens/mpi/2024/info-c

Prepa Help ne revendique aucun droit sur le sujet, propriété de l'organisateur du concours.

ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2024

JEUDI 18 AVRIL 2024
08h00 - 12h00

FILIERE MPI - Epreuve n° 7

INFORMATIQUE C (XULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Égalités et différences

Le sujet comporte 17 pages, numérotées de 1 à 17 et une annexe.

Vue d'ensemble du sujet.

Ce sujet traite du problème de satisfiabilité de formules construites par la conjonction de contraintes d'égalité et de différence (dis-égalité) entre expressions. Ce problème se présente, par exemple, dans la compilation de programmes, quand on cherche à réduire des expressions complexes (nécessitant beaucoup de calculs) vers des expressions plus simples, en utilisant les propriétés des opérateurs apparaissant dans ces expressions. Par exemple, si `x` est une variable entière, `(x * 2)/2 == x` est une contrainte d'égalité entre deux expressions écrites en C. De même, si `y` et `z` sont des variables booléennes, l'égalité `(y || (y && z)) == y` permet de simplifier l'expression C écrite à gauche de l'égalité.

Il est important de savoir si un ensemble donné de contraintes d'égalité ou de différence est satisfiable. Dans la suite, P_L ***est le problème suivant : « Étant donnée une formule*** F***, conjonction de contraintes de la classe*** L***, décider si*** F ***est satisfiable ».***

Ce sujet propose d'étudier la complexité du problème P_L et les algorithmes pour le résoudre dans le cas de quatre classes de contraintes :

La partie I considère la classe L_1 où les contraintes ont la forme $x \leftrightarrow y$ ou $x \leftrightarrow \neg z$ avec x, y et z des variables propositionnelles (c'est-à-dire, ayant des valeurs booléennes).

La partie II concerne la classe L_2 incluant les contraintes de la classe L_1 et celles de la forme $x \leftrightarrow (y \wedge z)$ (entre autres) avec x, y et z des variables propositionnelles. Les algorithmes des deux premières parties seront codés en OCaml.

La partie III s'intéresse à la classe L_3 des contraintes de la forme $x = y$ ou $x \neq y$ avec x et y des variables entières.

Enfin, la partie IV concerne les contraintes d'égalité et de différence entre les expressions construites à partir de variables entières et de fonctions à deux arguments. Les algorithmes des deux dernières parties seront codés en C.

Les parties III et IV peuvent être traitées indépendamment de parties I et II. Il est recommandé de traiter les parties I et II de manière séquentielle. De même pour les parties III et IV.

Notations

Les contraintes seront construites en utilisant un ensemble $\mathbb{X}$ de n variables notées X_i avec $1 \leq i \leq n$. Une valeur de vérité est un élément de $\mathbb{B} \stackrel{\text{def}}{=} \{0, 1\}$ où 1 représente le vrai et 0 le faux. Une valeur entière k est un élément de $\mathbb{Z}$ et sa valeur absolue est notée $|k|$. Étant donnés $k, m \in \mathbb{Z}$ avec $k \leq m$, l'intervalle $[k; m]$ est l'ensemble des $i \in \mathbb{Z}$ tels que $k \leq i \leq m$.

Logique propositionnelle. On utilisera les formules de la logique propositionnelle construites à partir de l'ensemble de variables propositionnelles $\mathbb{X}$, les constantes propositionnelles $\bot$ (faux) et $\top$ (vrai), les connecteurs logiques classiques $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction). On définit $F \to G$ (implication) par $(\neg F \vee G)$, et $F \leftrightarrow G$ (équivalence) par $(F \to G) \wedge (G \to F)$. Pour les connecteurs $\wedge$ et $\vee$, on utilise leur forme k-aire avec $k \geq 0$. Par exemple, $X_1 \wedge X_2 \wedge \ldots \wedge X_k$ est une formule utilisant une conjonction k-aire. Par convention, une conjonction 0-aire est $\top$, et une disjonction 0-aire est $\bot$; une conjonction (resp. disjonction) unaire est identique à son opérande.

Un ***littéral*** ℓ est soit une variable X_i soit sa négation $\neg X_i$. Une formule est en ***forme normale conjonctive*** (CNF) si c'est une conjonction de disjonctions de littéraux. Une formule est en forme k-CNF avec $k > 0$, si elle est en forme CNF et le nombre de littéraux différents dans chaque disjonction est au plus k. Par exemple, la formule $(X_2 \vee X_3 \vee \neg X_4) \wedge (\neg X_3 \vee X_4) \wedge X_1$ est en forme 3-CNF.

On rappelle qu'une formule propositionnelle F peut être ***satisfaite*** ou non par une valuation $v : \mathbb{X} \to \mathbb{B}$, qui assigne une valeur de vérité à chaque variable. On note $v \models F$ la relation de satisfaction. Une formule F' est ***conséquence logique*** de F, noté $F \models F'$, quand, pour toute valuation v telle que $v \models F$, on a aussi $v \models F'$. Deux formules sont ***logiquement équivalentes*** lorsque chacune est conséquence logique de l'autre (ou, de façon équivalente, quand elles sont satisfaites par les mêmes valuations). Une formule est une ***tautologie*** si elle est satisfaite par toutes les valuations. Une formule est une ***contradiction*** (antilogie) ou contradictoire si elle n'est satisfaite par aucune valuation. Une formule F est ***satisfiable*** s'il existe une valuation v telle que $v \models F$.

L'annexe rappelle certaines règles de la déduction naturelle.

Complexité. Par la complexité d'un algorithme A on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de A ***dans le pire cas***. Lorsque la complexité dépend d'un ou plusieurs paramètres $k_0, \ldots, k_r$, on dit que A a une complexité en $O(f(k_0, \ldots, k_r))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $k_0, \ldots, k_r$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $k_0, \ldots, k_r$ la complexité est au plus $C \times f(k_0, \ldots, k_r)$. De même, on dit que A a une complexité en $\Theta(f(k_0, \ldots, k_r))$ s'il existe deux constantes $C_1 > 0$ et $C_2 > 0$ telles que, pour toutes les valeurs de $k_0, \ldots, k_r$ suffisamment grandes, la complexité de A est au moins $C_1 \times f(k_0, \ldots, k_r)$ et au plus $C_2 \times f(k_0, \ldots, k_r)$. Dans tout ce sujet, on note « log » le logarithme à base 2, et on utilisera exclusivement ce logarithme.

Programmation. Dans les questions de programmation en C et OCaml, on n'utilise que les fonctions qui sont incluses dans la bibliothèque standard du langage.

Pour le code écrit en C, on suppose que les en-têtes `stdbool.h`, `stdio.h`, `stdlib.h` et `assert.h` ont été inclus.

On rappelle quelques opérations sur les listes en OCaml.

— `List.iter f lst` applique la fonction `f: 'a -> unit` à chaque élément de la liste `lst` de type `'a list`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.iter` est en $O(k)$, où k est la longueur de `lst`.

— `List.map f lst` renvoie la liste de type `'b list` obtenue en appliquant la fonction `f: 'a -> 'b` à chaque élément de la liste `lst` de type `'a list`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.map` est en $O(k)$, où k est la longueur de `lst`.

— `List.mem e lst` renvoie `true` si et seulement si `e` est un élément de `lst`. Si la complexité du test d'égalité est en $O(1)$, la complexité de `List.mem` est en $O(k)$, où k est la longueur de `lst`.

— `List.exists f lst` renvoie `true` si et seulement s'il existe au moins un élément `e` de `lst` (liste de type `'a list`) tel que `f e` renvoie `true` avec `f: 'a -> bool`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.exists` est en $O(k)$, où k est la longueur de `lst`.

— `List.for_all f lst` renvoie `true` si et seulement si tous les éléments de `lst` (de type `'a list`) satisfont le prédicat `f: 'a -> bool`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.for_all` est en $O(k)$, où k est la longueur de `lst`.

— `List.filter f lst` renvoie la liste de tous les éléments de `lst` (de type `'a list`) qui satisfont le prédicat `f: 'a -> bool`. Si la complexité de `f` est en $O(1)$, alors la complexité de `List.filter` est en $O(k)$, où k est la longueur de `lst`.

On rappelle quelques opérations sur les tableaux en OCaml.

— `Array.length tab` renvoie la longueur du tableau `tab` en temps $O(1)$.

— `Array.make k v` crée un tableau de `k` éléments, tous initialisés à `v`, en temps $O(\texttt{k})$.

— La valeur à la case numéro `i` du tableau `tab` est `tab.(i)`. Les cases sont numérotées à partir de zéro et l'accès est fait en temps $O(1)$.

— L'affectation de la case `i` du tableau `tab` avec la valeur `v` s'écrit `tab.(i) <- v`; sa complexité est en $O(1)$.

— `Array.copy tab` crée une copie du tableau `tab` en temps $O(k)$ avec k la longueur du tableau en argument.

Partie I. Équivalences de littéraux

Dans cette partie, on se place dans le cadre de la logique propositionnelle. On définit L_1 comme la classe des contraintes de la forme $\ell \leftrightarrow \ell'$ avec ℓ et ℓ' des littéraux. Une formule F est dite construite sur L_1, notation $F \in \mathcal{F}(L_1)$, si elle est une conjonction de contraintes de L_1. ***Dans cette partie, on considère que la variable X_1 est toujours interprétée à 1, donc qu'elle représente la valeur de vérité vrai.*** La formule F_0 définie ci-dessous est dans $\mathcal{F}(L_1)$:

$$F_0 \quad \stackrel{\text{def}}{=} \quad (X_3 \leftrightarrow \neg X_2) \wedge (X_4 \leftrightarrow X_3) \wedge (X_3 \leftrightarrow X_4) \wedge (X_1 \leftrightarrow \neg X_5)$$

Question 1. Donner une formule logiquement équivalente à F_0 en forme 2-CNF. ⌟

Question 2. Donner une réduction en temps linéaire du problème P_{L_1} au problème de satisfiabilité des formules en 2-CNF. ⌟

Mise en œuvre en OCaml. Une contrainte de L_1 est représentée par un couple d'entiers (i, j) avec $i, j \in [1; 2n]$. Pour tout $i \in [1; 2n]$, le littéral représenté par i, noté $lit(i)$, est X_i si $i \in [1; n]$ et $\neg X_{i-n}$ si $i \in [n+1; 2n]$. Par la convention sur l'interprétation de X_1, l'entier $i = 1$ (resp. $i = n+1$) représente la valeur de vérité 1 (resp. 0). Enfin, une formule dans $\mathcal{F}(L_1)$ est représentée par la liste des couples correspondant aux contraintes dont la formule est conjonction. Par convention, la liste vide représente $\top$. Le nombre de variables, n, est une variable globale entière. Ce codage est donné par les déclarations OCaml ci-dessous :

```
1 type contr1 = int * int   (* contraintes L1 *)
2 type eform1 = contr1 list (* formules L1 *)
3 exception Bad_literal of int
4 exception Exn_unsat
```

Les exceptions déclarées ci-dessus seront utilisées dans les fonctions de cette partie et de la partie suivante.

Question 3. Écrire les fonctions suivantes :

- `var_of_lit: int -> int` renvoie l'index de la variable apparaissant dans le littéral représenté par l'argument i. Par exemple, pour $n = 5$, la fonction renvoie 3 si l'argument est 3, et 2 si l'argument est 7. Si $i \notin [1; 2n]$ alors la fonction exécute `raise (Bad_literal i)`.
- `neg_lit: int -> int` renvoie, pour un argument i, l'entier j tel que $lit(j)$ est logiquement équivalent à $\neg lit(i)$; si $i \notin [1; 2n]$ alors la fonction exécute `raise (Bad_literal i)`.
- `eform1_is_wf: eform1 -> bool` renvoie `true` si et seulement si l'argument est « bien formé », c'est-à-dire qu'il contient uniquement des entiers qui représentent des littéraux sur l'ensemble de variables $\mathbb{X}$.

⌟

Dans la suite de ce sujet, on suppose que tous les entiers utilisés par les valeurs de type `contr1` et `eform1` représentent des littéraux sur l'ensemble de variables $\mathbb{X}$. Par conséquent, l'exception `Bad_literal` ne doit pas être levée.

Graphe d'une formule. Le ***graphe associé*** à $F \in \mathcal{F}(L_1)$, noté $G_F = (V_F, E_F)$, est un graphe non orienté dont les sommets V_F sont les entiers dans l'intervalle $[1; 2n]$. Les arêtes E_F sont les ensembles $\{i, j\}$ tels que $i, j \in V_F$, $i \neq j$ et F contient soit la contrainte $\mathit{lit}(i) \leftrightarrow \mathit{lit}(j)$, soit la contrainte $\mathit{lit}(i') \leftrightarrow \mathit{lit}(j')$ avec i' et j' égaux respectivement à `neg_lit`(i) et `neg_lit`(j). Par exemple, pour $n = 5$, la contrainte $(X_3 \leftrightarrow X_4)$ est représentée par les arêtes $\{3, 4\}$ et $\{8, 9\}$. On note que les contraintes de F de la forme $\ell \leftrightarrow \ell$ ne sont pas représentées par E_F.

Question 4. On suppose $n = 5$. Pour chacune des formules suivantes, donner le graphe associé :

$$F_1 \stackrel{\text{def}}{=} (X_3 \leftrightarrow X_4) \wedge (X_4 \leftrightarrow X_2) \wedge (X_3 \leftrightarrow \neg X_2) \wedge (X_5 \leftrightarrow X_2)$$
$$F_2 \stackrel{\text{def}}{=} (\neg X_4 \leftrightarrow X_2) \wedge (X_2 \leftrightarrow \neg X_3) \wedge (X_4 \leftrightarrow \neg X_5) \wedge (X_3 \leftrightarrow X_5)$$

⌟

On suppose disposer d'une bibliothèque munie d'un type `graph` et qui code des graphes non orientés. Les sommets sont représentés par des entiers. Les opérations suivantes sont disponibles pour la construction et l'exploration des graphes :

- `g_create : int -> graph` renvoie une valeur représentant un graphe sans arêtes et ayant k sommets, où k est donné en argument, supposé strictement positif. Les sommets sont numérotés de 1 à k. La complexité de cette opération est en $O(k)$.
- `g_add_edge : graph -> int -> int -> unit` ajoute au graphe donné comme premier argument l'arête entre les sommets donnés en 2ème et 3ème arguments. Si les sommets en argument ne sont pas déclarés dans le graphe ou si l'arête existe déjà, alors cette fonction ne change pas le graphe. Sinon, le graphe est modifié en place. La complexité de cette opération est en $O(k)$, avec k le nombre de sommets du graphe.
- `g_nb_vertex : graph -> int` renvoie le nombre de sommets du graphe en argument. La complexité de cette opération est en $O(1)$.
- `g_succ : graph -> int -> int list` renvoie la liste des sommets adjacents du sommet en second argument dans le graphe donné en premier argument. Si le sommet en argument n'existe pas, la liste renvoyée est vide. La complexité de cette opération est en $O(1)$.

Question 5. Écrire une fonction `eform1_to_graph: eform1 -> graph` qui renvoie le graphe associé à la formule F représentée par l'argument. Si F contient une contrainte de la forme $\ell \leftrightarrow \neg\ell$, alors la fonction lève l'exception **`Exn_unsat`**. ⌟

Lorsqu'un appel à `eform1_to_graph lst` lève l'exception **`Exn_unsat`**, la formule représentée par `lst` contient toujours une contrainte contradictoire. Cependant, une formule peut être une contradiction sans que l'exception soit levée, comme c'est le cas de la formule $(X_1 \leftrightarrow X_2) \wedge (X_2 \leftrightarrow \neg X_1)$.

Question 6. Donner et justifier soigneusement la complexité de `eform1_to_graph` en fonction de n et de la longueur p de la liste en argument. ⌟

Soient $F \in \mathcal{F}(L_1)$ une formule et `g` la représentation de G_F, le graphe associé à F. On note par $CC(G_F)$ l'ensemble des ensembles de sommets des composantes connexes de G_F. Soit m la taille de $CC(G_F)$, c'est-à-dire, le nombre de composantes connexes de G_F. On représente $CC(G_F)$ par un tableau `c` de $2n + 1$ entiers, qui associe à chaque `i` $\in [1; 2n]$, sommet de `g`, un entier dans l'intervalle $[1; m]$. Ainsi, `c.(i)` identifie de manière unique la composante connexe du sommet `i`. La case de `c` à la position 0 ne sera pas utilisée, mais elle permet d'utiliser directement les indices des variables comme positions de `c`. Par convention, `c.(0)` est 0.

Question 7. Écrire une fonction `g_cc_array: graph -> (int array * int)` qui renvoie le couple $(\texttt{c}, m)$, où `c` est le tableau des composantes connexes du graphe en argument et où m est le nombre de composantes connexes. ⌟

Question 8. Démontrer que si i et j sont des sommets distincts appartenant à la même composante connexe de G_F, alors $F \models \mathit{lit}(i) \leftrightarrow \mathit{lit}(j)$. ⌟

Question 9. Soit $C = \{i_1, \ldots, i_q\} \in CC(G_F)$ l'ensemble des sommets d'une composante connexe de G_F. Soit C' l'ensemble de sommets $\{j_1, \ldots, j_q\}$ avec $j_k = \texttt{neg_lit}(i_k)$ pour tout $k \in [1; q]$. Démontrer que C' est élément de $CC(G_F)$. ⌟

Question 10. Démontrer que F est une contradiction si et seulement s'il existe une composante connexe C de G_F et un sommet i de C tels que $\texttt{neg_lit}(i)$ est un sommet de C. ⌟

Question 11. Écrire une fonction `cc_is_sat: int array -> bool` qui a comme argument `c`, le tableau représentant les composantes connexes de G_F et qui renvoie `true` si et seulement si F est satisfiable. ⌟

Représentation d'une valuation. Une valuation v de $\mathbb{X}$ est représentée par un tableau `v` de $n+1$ valeurs 0 ou 1 tel que `v.(`i`)` est 1 si et seulement si $v(X_i) = 1$. En particulier, `v.(1)=1` par convention sur X_1. La case à la position `0` n'est pas utilisée, mais elle permet d'utiliser les indices des variables comme positions du tableau ; par convention, `v.(0)` est 0.

Question 12. Écrire une fonction `eform1_sat: eform1 -> (int array) option` qui a comme argument une liste `lst` représentant une formule $F \in \mathcal{F}(L_1)$. L'appel `eform1_sat lst` renvoie `None` si F est une contradiction. Si F est satisfiable, l'appel renvoie `Some(v)` avec `v` un tableau qui représente une valuation qui satisfait F. La complexité en temps de l'algorithme doit être en $O(p \times n)$ où p est la longueur de la liste `lst` en argument. Justifier que votre algorithme a cette complexité en temps. ⌟

Partie II. Équivalences avec une conjonction de deux littéraux

On reste dans le cadre de la logique propositionnelle, mais on considère une nouvelle classe de contraintes. La classe L_2 contient les contraintes de la forme $\ell \leftrightarrow (\ell' \wedge \ell'')$ où ℓ, ℓ', ℓ'' sont des littéraux. Pour identifier les contraintes des classes L_1 et L_2, on appelle ***i*-contrainte** une contrainte de la classe L_i avec $i \in \{1, 2\}$. Une formule F est dite construite sur L_2, notation $F \in \mathcal{F}(L_2)$, si elle est une conjonction de 2-contraintes. ***On garde l'hypothèse que*** X_1 ***est toujours interprétée à*** 1**.** Un exemple de formule de $\mathcal{F}(L_2)$ est :

$$F_3 \quad \stackrel{\text{def}}{=} \quad (X_1 \leftrightarrow (X_2 \wedge \neg X_4)) \wedge (\neg X_2 \leftrightarrow (X_2 \wedge X_3)) \wedge (X_4 \leftrightarrow (\neg X_1 \wedge X_5))$$

Question 13. Donner une réduction en temps polynomial du problème P_{L_2} au problème de satisfiabilité des formules en 3-CNF. ⌟

Question 14. Donner une dérivation formelle (preuve en déduction naturelle) du séquent suivant :

$$X_2 \to (\neg X_2 \wedge X_3), (\neg X_2 \wedge X_3) \to X_2 \quad \vdash \quad \neg X_2.$$

Les règles de la déduction naturelle à utiliser dans cette preuve sont rappelées dans l'annexe. ⌟

En utilisant la convention sur l'interprétation de X_1, toute 1-contrainte $\ell \leftrightarrow \ell'$ est logiquement équivalente à la 2-contrainte $\ell \leftrightarrow (\ell' \wedge X_1)$. À l'inverse, certaines 2-contraintes sont logiquement équivalentes à des formules de $\mathcal{F}(L_1)$. Par exemple, les équivalences logiques suivantes sont vraies pour tout couple de littéraux ℓ et ℓ' (les preuves ne sont pas demandées) :

$\ell \leftrightarrow (X_1 \wedge \ell')$ équivalent à $\ell \leftrightarrow \ell'$	par 1 élément neutre de $\wedge$	(1)
$\ell \leftrightarrow (\neg X_1 \wedge \ell')$ équivalent à $\ell \leftrightarrow \neg X_1$	par 0 élément absorbant de $\wedge$	(2)
$\ell \leftrightarrow (\ell' \wedge \ell')$ équivalent à $\ell \leftrightarrow \ell'$	par idempotence de $\wedge$	(3)
$\ell \leftrightarrow (\ell' \wedge \neg\ell')$ équivalent à $\ell \leftrightarrow \neg X_1$	par contradiction	(4)
$X_1 \leftrightarrow (\ell \wedge \ell')$ équivalent à $(X_1 \leftrightarrow \ell) \wedge (X_1 \leftrightarrow \ell')$	par définition de $\wedge$	(5)
$\ell \leftrightarrow (\neg\ell \wedge \ell')$ équivalent à $(\neg X_1 \leftrightarrow \ell) \wedge (\neg X_1 \leftrightarrow \ell')$	admis	(6)

Dans la suite de cette partie, on dénote par $\mathbb{E}(\mathbb{X})$ toutes les équivalences définies sur l'ensemble des littéraux par les équivalences de (1) à (6) et par leur variantes obtenues en utilisant la symétrie de la conjonction. Par exemple, « $\ell \leftrightarrow (\ell' \wedge X_1)$ équivalent à $\ell \leftrightarrow \ell'$ » est également une équivalence de $\mathbb{E}(\mathbb{X})$, grâce à la symétrie de la conjonction appliquée à (1).

Mise en œuvre en OCaml. Une 2-contrainte $\ell \leftrightarrow (\ell' \wedge \ell'')$ est représentée par un enregistrement avec trois champs entiers. Les entiers de ces enregistrements prennent des valeurs dans $[1; 2n]$ et représentent des littéraux avec la même convention de codage qu'en première partie. Une formule dans $\mathcal{F}(L_2)$ est représentée par une liste d'enregistrements ; la liste vide représente $\top$. Ce codage est donné par les types OCaml suivants :

```
1 type contr2 = { ll: int; lr1: int; lr2: int } (* contraintes L2 *)
2 type eform2 = contr2 list                      (* formules L2 *)
```

Une valeur `e` de type `contr2` représente la 2-contrainte `e.ll` $\leftrightarrow$ (`e.lr1` $\wedge$ `e.lr2`).

Dans la suite de cette partie, on suppose que toutes les valeurs des types `contr`i et `eform`i avec $i \in \{1, 2\}$ sont bien formées, c'est-à-dire que les entiers qu'elles contiennent représentent des littéraux sur $\mathbb{X}$.

Question 15. Écrire la fonction `contr2_simplify_eq5: contr2 -> eform1 option` qui applique l'équivalence (5) pour simplifier la 2-contrainte en argument. L'appel `contr2_simplify_eq5 e2` :

- renvoie **`None`** si la simplification ne peut pas être faite avec (5) ;
- lève l'exception **`Exn_unsat`** (déclarée en partie I) si la simplification peut être faite mais la 2-contrainte représentée par `e2` est une contradiction ;
- enfin, renvoie **`Some`** `[(1,e2.lr1);(1,e2.lr2)]` si la simplification peut être effectuée sans détecter de contradiction. ⌟

Valuation partielle. Une valuation partielle est une fonction $v : \mathbb{X} \to \{-1, 0, 1\}$ telle que $v(X_1) = 1$. Pour tout $i \neq 1$, si $v(X_i) = b \in \mathbb{B}$ alors on dit que v assigne X_i à b. Si $v(X_i) = -1$ alors on dit que X_i n'est pas assignée par v. On étend la notion de valuation partielle aux littéraux et aux formules comme suit :

- $v(\neg X_i) = \neg b$ si $v(X_i) = b$ avec $b \in \mathbb{B}$, et $v(\neg X_i) = -1$ si $v(X_i) = -1$.
- Soit la 1-contrainte $\ell \leftrightarrow \ell'$. Si $v(\ell), v(\ell') \in \mathbb{B}$ alors $v(\ell \leftrightarrow \ell') = 1$ si $v(\ell) = v(\ell')$ et 0 sinon. Si au moins une des valeurs $v(\ell)$ ou $v(\ell')$ est -1, alors $v(\ell \leftrightarrow \ell') = -1$.
- Pour la conjonction de deux littéraux $\ell \wedge \ell'$, $v(\ell \wedge \ell') = -1$ si au moins une des valeurs $v(\ell)$ ou $v(\ell')$ est -1 ; sinon la valeur de $v(\ell \wedge \ell')$ est 1 si et seulement si $v(\ell) = 1$ et $v(\ell') = 1$.
- Soit la 2-contrainte $\ell \leftrightarrow (\ell' \wedge \ell'')$. Si $v(\ell), v(\ell' \wedge \ell'') \in \mathbb{B}$ alors $v(\ell \leftrightarrow (\ell' \wedge \ell'')) = 1$ si $v(\ell) = v(\ell' \wedge \ell'')$ et 0 sinon. Si au moins une des valeurs $v(\ell)$ ou $v(\ell' \wedge \ell'')$ est -1, alors $v(\ell \leftrightarrow (\ell' \wedge \ell'')) = -1$.

La ***valeur d'une formule*** $F_i \in \mathcal{F}(L_i)$ où $i \in \{1, 2\}$ pour une valuation partielle v, notation $v(F_i)$, est -1 s'il existe une contrainte e de F_i telle que $v(e) = -1$; sinon, $v(F_i) = 1$ si pour toute contrainte e de F_i on a $v(e) = 1$, et $v(F_i) = 0$ sinon.

Une valuation partielle v est ***compatible*** avec une valuation v' si pour tout $X_i \in \mathbb{X}$ tel que $v(X_i) \in \mathbb{B}$ on a $v'(X_i) = v(X_i)$.

Une valuation partielle v est représentée en OCaml par un tableau `v` de taille $n + 1$ dont les éléments appartiennent à $\{-1, 0, 1\}$. Comme pour les valuations, la case de position `0` n'est pas utilisée et, par convention, `v.(0)=0`.

Question 16. Écrire la fonction `eform1_of_valp: int array -> eform1` qui renvoie une formule F de $\mathcal{F}(L_1)$ telle que F encode les assignations de la valuation partielle `v` en argument. Ainsi, un appel à `eform1_of_valp v` renvoie la liste de couples (i, j) tels que $i > 0$, `v.(`i`)` $\neq -1$ et si `v.(`i`)` $= 1$ alors $j = 1$ et si `v.(`i`)` $= 0$ alors $j = n + 1$.

Soit F la formule représentée par le résultat de l'appel à `eform1_of_valp v`. Démontrez que, pour toute valuation v' on a $v' \models F$ si et seulement si `v` est compatible avec v'. ⌟

Une 2-contrainte peut être simplifiée en présence d'une valuation partielle v en une formule de $\mathcal{F}(L_1)$. Par exemple, si $v(X_2) = 1$ alors la 2-contrainte $X_2 \leftrightarrow (\ell' \wedge \ell'')$ a la même valeur que la 1-contrainte $(X_1 \leftrightarrow \ell') \wedge (X_1 \leftrightarrow \ell'')$ pour la valuation partielle v. On suppose donnée une fonction `contr2_simplify_valp: contr2 -> int array -> eform1 option` qui applique les équivalences de $\mathbb{E}(\mathbb{X})$ pour simplifier la 2-contrainte `e2` en premier argument en tenant compte de

la valuation partielle `vp` donnée comme second argument. Plus précisement, si `vp(e2)=0`, alors la fonction lève l'exception **`Exn_unsat`**. Si aucune simplification ne s'applique, alors le résultat est **`None`**. Si `e2` est simplifiable, le résultat est **`Some(lst)`** avec `lst` une liste représentant une formule F_1' de $\mathcal{F}(L_1)$ obtenue par simplification de `e2` telle que `vp(e2) = vp(`F_1'`)`. La complexité de `contr2_simplify_valp` est en $O(1)$.

Algorithme. Les questions suivantes vous guident pour prouver la correction et la terminaison de l'algorithme de décision pour P_{L_2} qui est mis en œuvre dans la fonction `eform2_sat` de la figure 1.

On cherche à démontrer que la fonction `eform2_sat` est correcte, c'est-à-dire qu'un appel à `eform2_sat lst0` doit renvoyer **`None`** si la formule $F_0 \in \mathcal{F}(L_2)$ représentée par `lst0` est une contradiction. Sinon, l'appel doit renvoyer **`Some(v)`** où `v` est un tableau qui représente une valuation qui satisfait F_0. L'algorithme mis en œuvre pour `eform2_sat` est un algorithme de type retour sur trace (***backtracking***) dont le principe est de réduire la formule $F_0 \in \mathcal{F}(L_2)$ en une formule de $\mathcal{F}(L_1)$ afin d'appliquer `eform1_sat`, l'algorithme polynomial de la partie I. Pour effectuer cette réduction, `eform2_sat` appelle à la ligne 5 la fonction récursive `eform2_sat_aux` dont les arguments sont une liste `l2` représentant une formule $F_2 \in \mathcal{F}(L_2)$, une liste `l1` représentant une formule $F_1 \in \mathcal{F}(L_1)$ et une valuation partielle `vp`.

La correction de `eform2_sat_aux` est un lemme important de la preuve de correction de `eform2_sat`. Un appel à `eform2_sat_aux l2 l1 vp` est correct s'il renvoie un tableau qui représente une valuation v telle que $v \models F_2 \wedge F_1$ et `vp` est compatible avec v, si une telle valuation existe ; sinon, l'appel doit lever l'exception **`Exn_unsat`**.

La mise en œuvre de `eform2_sat_aux` cherche à compléter la valuation partielle `vp` en assignant des valeurs à certaines variables (non assignées) qui apparaissent dans les littéraux de F_2 et qui permettent de simplifier F_2 en une formule de $\mathcal{F}(L_1)$. Pour savoir si une 2-contrainte `e2` de `l2` peut être simplifiée en présence de `vp`, on appelle `contr2_simplify_valp e2 vp` (ligne 15). Si `e2` se simplifie, l'argument `l1` accumule les formules de $\mathcal{F}(L_1)$ obtenues par la simplification et `eform2_sat_aux` continue la simplification du reste des contraintes de `l2` (ligne 17). Si la contrainte `e2` est une contradiction en présence de `vp`, l'exception **`Exn_unsat`** est levée par `contr2_simplify_valp e2 vp`. Si `e2` ne peut pas être simplifiée (ligne 19), alors `eform2_sat_aux` explore les simplifications de la contrainte `e2` obtenues en assignant la variable du littéral `e2.lr1` à chaque valeur de $\mathbb{B}$. L'exploration d'une assignation est faite grâce à un appel récursif à `eform2_sat_aux` avec le paramètre représentant la valuation partielle mis à jour avec l'assignation choisie.

Quand toutes les 2-contraintes de F_2 ont été simplifiées, c'est-à-dire quand la liste `l2` est vide (ligne 10), `eform2_sat_aux` appelle `eform1_sat` avec l'argument `(l1'@l1)` où la liste `l1'` est obtenue par l'appel à `eform1_of_valp vp`. Si l'appel à `eform1_sat (l1'@l1)` renvoie `Some(v)`, alors le résultat de `eform2_sat_aux` est `v`. Sinon, `eform2_sat_aux` lève l'exception **`Exn_unsat`**. Ceci a comme effet de revenir dans un appel précédent (sur la pile d'appels) de `eform2_sat_aux` pour explorer une autre assignation (lignes 25–28) ou, si toutes les assignations ont été explorées, de terminer l'appel à `eform2_sat_aux l2 l1 vp` par l'exception **`Exn_unsat`**.

Question 17. Démontrer que tout appel `eform2_sat lst0` termine. ⌟

Question 18. Démontrer que la fonction `eform2_sat` est correcte en supposant que la fonction `eform2_sat_aux` est correcte. ⌟

Dans les trois questions suivantes, on cherche à démontrer les principaux cas de la correction de `eform2_sat_aux`.

```
1  let rec eform2_sat (lst:eform2) : (int array) option =
2    try
3      let vinit = Array.make (n+1) (-1) in (
4        vinit.(0) <- 0; vinit.(1) <- 1;
5        Some (eform2_sat_aux lst [] vinit) )
6    with Exn_unsat -> None
7
8  and eform2_sat_aux (l2:eform2) (l1:eform1) (vp:int array) : int array =
9    match l2 with
10   | [] -> let l1' = eform1_of_valp vp in
11           let  r  = eform1_sat (l1'@l1) in
12           (match r with None -> raise Exn_unsat | Some v -> v)
13
14   | e2::l2' ->
15     match contr2_simplify_valp e2 vp with
16     | Some l1' -> (* simplification *)
17         eform2_sat_aux l2' (l1'@l1) vp
18
19     | None -> (* branchement sur valeur de e2.lr1 *)
20       let xlr1 = var_of_lit e2.lr1 in
21       try
22         let vp1 = Array.copy vp in (
23           vp1.(xlr1) <- 1;
24           eform2_sat_aux l2 l1 vp1 )
25       with Exn_unsat ->
26         let vp0 = Array.copy vp in (
27           vp0.(xlr1) <- 0;
28           eform2_sat_aux l2 l1 vp0 )
```

FIGURE 1 – Mise en œuvre OCaml de l'algorithme de décision pour P_{L_2}

Question 19. Démontrez la correction de `eform2_sat_aux` quand son premier argument est la liste vide, on supposant que les fonctions `eform1_of_valp` et `eform1_sat` sont correctes. ⌟

Question 20. Soit un appel `eform2_sat_aux l2 l1 vp` tel que `l2` est `e2::l2'` et `contr2_simplify_vp e2 vp` renvoie `Some l1'`. Démontrez la correction d'un tel appel en supposant que l'appel récursif à `eform2_sat_aux` de la ligne 17 est correct. ⌟

Question 21. Soit un appel `eform2_sat_aux l2 l1 vp` tel que `l2` est `e2::l2'` et `contr2_simplify_vp e2 vp` renvoie `None`. Démontrez la correction d'un tel appel en supposant que les appels récursifs aux lignes 24 et 28 sont corrects. ⌟

Question 22. Donner la complexité de `eform2_sat` en fonction de la longueur k de la liste en argument et du nombre de variables n. ⌟

Partie III. Égalités et différences entre entiers

Dans cette partie, le domaine d'interprétation des variables est $\mathbb{Z}$**.** La classe L_3 est constituée de contraintes de la forme $X_i = X_j$ et $X_i \neq X_j$, avec $X_i, X_j \in \mathbb{X}$. Une formule F est dite construite sur L_3, notation $F \in \mathcal{F}(L_3)$, si elle est une conjonction de contraintes de L_3. Une valuation est une fonction $v : \mathbb{X} \to \mathbb{Z}$ qui assigne une valeur dans $\mathbb{Z}$ à chaque variable. Une valuation v ***satisfait*** une contrainte $X_i = X_j$ (resp. $X_i \neq X_j$) si et seulement si $v(X_i) = v(X_j)$ (resp. $v(X_i) \neq v(X_j)$). Une valuation v ***satisfait*** une formule $F \in \mathcal{F}(L_3)$ si et seulement si v satisfait chaque contrainte de F. Les notions de conséquence logique, équivalence logique, tautologie et antilogie sont définies de manière similaire à la logique propositionnelle.

Le problème P_{L_3} peut être réduit (en temps polynomial en n et en la taille de la formule) au problème de satisfiabilité d'une formule propositionnelle. Dans cette partie, on étudie un algorithme polynomial en n et en la taille de la formule pour P_{L_3}.

Mise en œuvre en langage C. On propose de représenter les formules de $\mathcal{F}(L_3)$ par des listes simplement chaînées acycliques de type `eform3_t*`. Chaque cellule `c` (de type `eform3_t`) d'une telle liste représente une contrainte $e \in L_3$ grâce à trois données : un booléen `c.is_eq` qui est vrai si et seulement si e est une égalité, et deux entiers positifs `c.xi` et `c.xj` qui représentent les indices des variables utilisées par e. La formule $\top$ est représentée par la liste vide de valeur `NULL`. On suppose que le nombre n de variables dans $\mathbb{X}$ est déclaré comme une variable globale entière. Enfin, on suppose que toute structure `eform3_t` est bien formée, c'est-à-dire que tous les entiers qu'elle contient appartiennent à l'intervalle $[1; n]$.

Question 23. Donner la déclaration du type `eform3_t`. ⌟

Question 24. Écrire la fonction `void eform3_print(eform3_t *lst)` qui affiche sur la sortie standard chaque contrainte (par exemple, sous la forme `X_1 != X_4`) de la formule représentée par `lst`, une contrainte par ligne ; si `lst` est `NULL`, la fonction affiche `true` sur une ligne. ⌟

Classes d'équivalence avec « unir et trouver ». On utilise une structure unir et trouver (***union-find***) pour résoudre le problème P_{L_3}. On suppose disposer d'une bibliothèque C qui fournit une implémentation d'une structure unir et trouver sur des ensembles d'entiers. Cette bibliothèque exporte les déclarations de types et de fonctions suivantes :

```
1  /* Rappel: definition dans stdlib.h
2  typedef unsigned int size_t; */
3
4  struct uf_s {      /* type unir et trouver (union-find) */
5      size_t nelem;  /* nombre d'elements de l'ensemble */
6      int   *parent; /* tableau de taille nelem = relation parent */
7      ...            /* champs utiles pour operations en O(log n) */
8  };
9  typedef struct uf_s uf_t;
10 uf_t *uf_create(size_t sz);
11 int   uf_find(uf_t *cs, int i);
12 void  uf_union(uf_t *cs, int i1, int i2);
```

Dans ce qui suit, on supposera les propriétés suivantes de cette implémentation :

— `uf_create(s)` renvoie un pointeur `cs` vers une structure unir et trouver allouée dans le tas et qui représente une partition de l'ensemble $\{0,\ldots,\mathtt{s}-1\}$ où chaque élément de l'ensemble est seul dans sa classe d'équivalence ; `s` donne la valeur du champ `cs->nelem`.

— `uf_find(cs,i)` renvoie la classe de `i`, c'est-à-dire un entier entre 0 et `cs->nelem-1` qui est le représentant de cette classe.

— `uf_union(cs,i1,i2)` fusionne les classes de `i1` et `i2` en modifiant la structure `*cs`.

Si `cs` est un pointeur valide vers une structure `uf_t` avec `cs->nelem==s`, le champ `cs->parent` pointe vers un tableau de `s` entiers ayant des valeurs dans l'intervalle $[0;\mathtt{s}-1]$; ce tableau encode la relation représentant d'une classe d'équivalence. Si `cs->parent[i]==j` alors `i` est dans la classe de `j`. Le tableau `cs->parent` est initialisé par `uf_create` de sorte que `cs->parent[i]==i`. Les opérations ci-dessus sur la structure unir et trouver ont une complexité en $\Theta(\mathtt{s})$ pour `uf_create` et en $O(\log \mathtt{s})$ pour `uf_find` et `uf_union`.

Pour notre problème, ***l'ensemble de valeurs à partitionner sera*** $[0;n]$, c'est-à-dire l'ensemble d'indices de variables dans $\mathbb{X}$ auquel on ajoute la valeur 0. La valeur 0 ne sera pas utilisée, mais elle permet d'utiliser directement les indices des variables comme positions du tableau `cs->parent`.

Question 25. Écrire la fonction `uf_t *eform3_to_uf(eform3_t *lst)` qui renvoie une valeur de type `uf_t*` représentant les classes d'équivalence sur $[0;n]$ définies par les ***contraintes d'égalité*** de la formule représentée par `lst`. On suppose que `lst` n'est pas vide. Les contraintes de différence de `lst` seront ignorées dans cette fonction. ⌟

Algorithme de décision. D'après la question précédente, la structure unir et trouver permet donc de représenter les contraintes d'égalité d'une formule de $\mathcal{F}(L_3)$, mais pas les contraintes de différence.

Question 26. Écrire une fonction `bool eform3_sat_int(eform3_t *lst)` qui renvoie `true` si et seulement si toutes les contraintes de différence de la formule représentée par `lst` peuvent être satisfaites en respectant les contraintes d'égalité de `lst`. Si `lst` est `NULL`, la fonction renvoie `true`. Justifier la correction de la fonction `eform3_sat_int`, c'est-à-dire, que tout appel à cette fonction renvoie `true` si et seulement si la formule représentée par son argument est satisfiable. ⌟

Question 27. On suppose que `lst` (de type `eform3_t*`) représente une formule avec un nombre de contraintes d'égalité égal à e et un nombre de contraintes de différence égal à d. Donner la complexité de l'appel à `eform3_sat_int(lst)` en fonction de n, e et d. ⌟

Calculer une valuation. Dans la suite de cette partie, on se propose de calculer, si elle existe, une valuation $v : \mathbb{X} \to \mathbb{Z}$ qui satisfait une formule de $\mathcal{F}(L_3)$.

Question 28. Écrire une fonction `int uf_classes(uf_t *cs)` qui renvoie le nombre de classes d'équivalence de la structure unir et trouver pointée par `cs` ; si `cs` est `NULL` alors la fonction renvoie 0. Cette fonction doit avoir une complexité en $O(n)$. ⌟

Question 29. Écrire une fonction `int *uf_valuation(uf_t *cs)` qui renvoie un pointeur `v` vers un tableau de $n+1$ entiers, alloué dans le tas. Soit k le résultat de l'appel de `uf_classes(cs)`. On suppose $k \geq 2$. Pour tout $i \in [1;n]$, `v[`i`]` est une valeur entière dans l'intervalle $[1;k]$ qui

est associée ***de façon unique*** à (chaque élément de) la classe d'équivalence de i. La case à la position 0 de `v` n'est pas utilisée, mais elle permet d'utiliser les indices des variables comme positions de `v` ; par convention, `v[0]==0`. La fonction `uf_valuation` doit avoir une complexité en $O(n \log n)$. ⌟

Soit $F \in \mathcal{F}(L_3)$ une formule représentée par une liste `lst` différente de `NULL` telle que `eform3_sat_int(lst)` renvoie `true`. Soient `cs` le résultat de l'appel à `eform3_to_uf(lst)` et `v` le résultat de l'appel à `uf_valuation(cs)`.

Question 30. Démontrer que `v` pointe vers un tableau qui représente une valuation qui satisfait la formule F représentée par `lst`. ⌟

Question 31. Si F ne contient pas de contraintes de différence, alors quel est le nombre minimal de valeurs que peut avoir l'image d'une valuation v qui satisfait F ? Donner un exemple d'une telle valuation.

La même question si F contient une seule contrainte de différence. ⌟

Question 32. Lorsque cela est possible, on souhaite construire une valuation v telle que v satisfait la formule F et l'image de v est un ensemble de taille 2. Quel algorithme sur les graphes peut être utilisé pour obtenir ce résultat et sur quel graphe ? Vous devez définir formellement le graphe en entrée de l'algorithme et donner la complexité de l'algorithme. ⌟

Partie IV. Égalités et différences d'expressions

Soit $\mathbb{F}$ un ensemble de m symboles de fonction notés f_i $(1 \leq i \leq m)$. Chaque symbole $f_i \in \mathbb{F}$ représente une fonction de $\mathbb{Z} \times \mathbb{Z}$ dans $\mathbb{Z}$. Comme dans la partie III, on interprète les variables X_i de $\mathbb{X}$ dans l'ensemble $\mathbb{Z}$. Les symboles de fonctions de $\mathbb{F}$ ***ne sont pas interprétés***, c'est-à-dire qu'ils n'ont pas de définition. Les seules propriétés connues de ces fonctions sont données sous la forme de contraintes d'égalité ou de différence. Ces contraintes sont celles de la classe L_4 définie ci-dessous.

Les contraintes de la classe L_4 utilisent des termes. Un ***terme*** t est défini de manière inductive : t est

— soit X_i, une variable de $\mathbb{X}$,

— soit $f_i(t_1, t_2)$, l'application d'une fonction $f_i \in \mathbb{F}$ sur deux termes t_1 et t_2, appelés aussi ***sous-termes directs*** de t.

Le ***symbole racine*** du terme X_i (resp. $f_j(t_1, t_2)$) est X_i (resp. f_j). On note $\mathbb{T}(\mathbb{X}, \mathbb{F})$ l'ensemble des termes construits avec les variables de $\mathbb{X}$ et les symboles de fonctions $\mathbb{F}$.

Ces termes peuvent être représentés par des arbres binaires. La figure 2 représente les arbres binaires correspondants aux termes :

$$t_1 \stackrel{\text{def}}{=} f_1(f_2(X_1, X_2), X_2) \quad \text{et} \quad t_2 \stackrel{\text{def}}{=} f_3(X_1, f_2(X_1, X_2)).$$

Chaque noeud de ces arbres est étiqueté en partie haute par un nom unique u_i appelé identifiant, et en partie basse par le symbole racine du terme. L'arc à gauche (resp. droite) d'un noeud correspond au premier (resp. second) sous-terme direct et est dessiné avec un trait en pointillé (resp. plein).

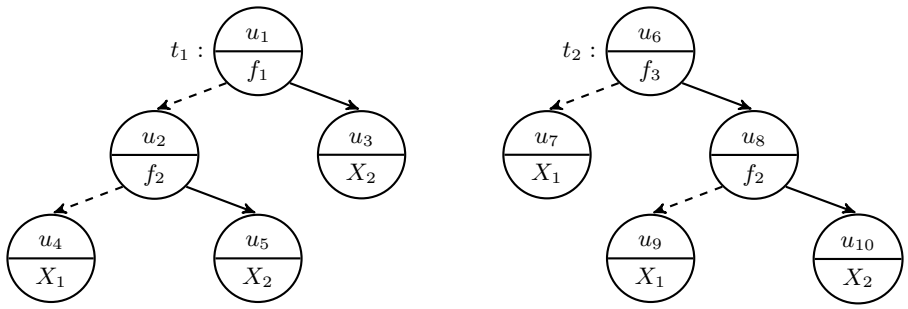


FIGURE 2 – Arbres des termes t_1 (à gauche) et t_2 (à droite)

L'ensemble des sous-termes d'un terme t, noté $\mathcal{T}(t)$, est défini de manière inductive par

$$\mathcal{T}(X_i) \stackrel{\text{def}}{=} \{X_i\} \quad \text{et} \quad \mathcal{T}(f_i(t_1, t_2)) \stackrel{\text{def}}{=} \{f_i(t_1, t_2)\} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2).$$

Question 33. Donner $\mathcal{T}(t_1)$ et $\mathcal{T}(t_2)$ pour les termes t_1 et t_2 définis ci-dessus. ⌟

La classe L_4 contient les contraintes de la forme $t_1 = t_2$ ou $t_1 \neq t_2$ avec t_1 et t_2 des termes. Une formule F de $\mathcal{F}(L_4)$ est une conjonction de contraintes L_4. L'ensemble des sous-termes d'une contrainte e de la forme $t_1 = t_2$ ou $t_1 \neq t_2$ est $\mathcal{T}(e) \stackrel{\text{def}}{=} \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$. L'ensemble des sous-termes d'une formule F de $\mathcal{F}(L_4)$, noté $\mathcal{T}(F)$, est défini comme l'union des ensembles des sous-termes de toutes les contraintes de F.

Mise en œuvre en langage C. Afin de concevoir un algorithme de décision pour $P(L_4)$, nous représentons ***les contraintes d'égalité*** d'une formule de $\mathcal{F}(L_4)$ par une structure de données appelée ***e-graphe***. Cette structure combine un graphe orienté acyclique avec une structure unir et trouver. Les définitions de types C suivantes codent une structure de e-graphe :

```
1  struct tnode_s {  /* type des noeuds des arbres representant les termes */
2      char *label;  /* symbole racine */
3      int   li;     /* position dans tset du premier sous-terme direct ou 0 */
4      int   ri;     /* position dans tset du second sous-terme direct ou 0 */
5  };
6  typedef struct tnode_s tnode_t;
7
8  struct egraph_s {     /* type e-graphe representant une formule L4 */
9      size_t   tset_sz; /* longueur de tset */
10     tnode_t *tset;    /* debut tableau des termes de la formule */
11     int     *pre;     /* relation sous-terme -- terme */
12     uf_t    *uf;      /* unir et trouver pour les classes des termes */
13 };
14 typedef struct egraph_s egraph_t;
```

Dans ce qui suit, on supposera les propriétés suivantes de la mise en œuvre de la structure `egraph_t`. Si `g` est une variable de type `egraph_t` représentant les contraintes d'égalité d'une formule $F \in \mathcal{F}(L_4)$, `g.tset` code l'ensemble des sous-termes $\mathcal{T}(F)$ comme suit :

- `g.tset` est un pointeur vers le debut d'un tableau de taille `g.tset_sz` ;
- `g.tset[0]` est initialisée à la valeur `{.label=NULL, .li=0, .ri=0}` (premier champ à `NULL` et les suivants à 0) de `tnode_t`, valeur qu'on appelle aussi ***terme vide*** ;
- un terme est représenté par une valeur `t` de type `tnode_t` tel que `t.label` est une chaîne de caractères qui décrit le symbole racine du terme, et `t.li` (resp. `t.ri`) est la position du premier (resp. second) sous-terme direct dans le tableau `g.tset`.

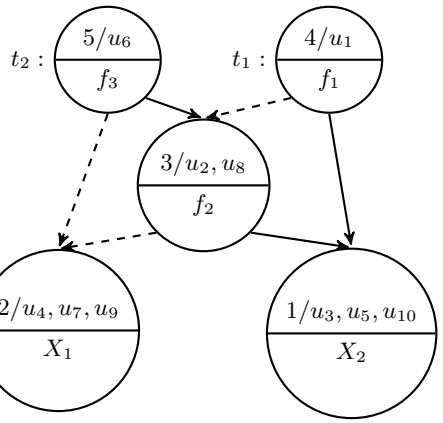
Un invariant du tableau `g.tset` est le suivant :

> ***tout terme de $\mathcal{T}(F)$ est représenté par une unique position du tableau*** `g.tset`.

Cet invariant correspond à une factorisation des termes communs et donc à la transformation de l'ensemble des arbres correspondant aux termes en un graphe orienté acyclique.

Par exemple, considérons la formule $t_1 = t_2$ utilisant les termes représentés dans la figure 2. La partie droite de la figure 3 représente le tableau `g.tset` de l'ensemble des sous-termes de la formule (une valeur `tnode_t` par ligne) et le terme vide. La partie gauche de la figure 3 représente le tableau `g.tset` sous la forme d'un graphe orienté acyclique : les noeuds contiennent en partie haute la position du terme dans le tableau `g.tset` et, pour la lisibilité de la transformation, l'ensemble des identifiants u_i des nœuds correspondants dans les arbres de la figure 2. La partie basse d'un noeud donne le symbole racine.

La composante `g.pre` de l'e-graphe contient la relation prédécesseur dans le graphe dirigé acyclique représenté par `g.tset`. Cette relation est donnée sous la forme d'une matrice linéarisée en un tableau de `g.tset_sz`^{2} valeurs dans l'ensemble $\{0, 1\}$. Si `g.pre[i*g.tset_sz + j]` vaut 1 alors le terme `g.tset[i]` a comme sous-terme direct le terme `g.tset[j]`. La valeur 0 dénote que `g.tset[j]` n'est pas sous-terme direct de `g.tset[i]`.



0 :	`NULL`	0	0
1 :	`"X2"`	0	0
2 :	`"X1"`	0	0
3 :	`"f2"`	2	1
4 :	`"f1"`	3	1
5 :	`"f3"`	2	3

FIGURE 3 – Codage de $\mathcal{T}(t_1 = t_2)$ sous forme de graphe (à gauche) et de tableau (à droite)

La composante `g.uf` de l'e-graphe `g` code les classes d'équivalence définies par les contraintes d'égalités entre les termes. La définition du type `uf_t` est la même que celle donnée dans la partie III. Chaque terme est représenté dans l'ensemble des éléments de `g.uf` par la position du noeud racine du terme dans le tableau `g.tset`. Par conséquent, un autre invariant de la structure d'e-graphe est le suivant :

> **`g.uf->nelem` *est égal à* `g.tset_sz`.**

Question 34. Écrire la fonction `bool term_is_diff(egraph_t *g, int tid1, int tid2)` qui teste si la contrainte $t_1 \neq t_2$ peut être satisfaite simultanément aux contraintes d'égalité représentées par `g`. Les paramètres `tid1` et `tid2` sont les positions dans `g.tset` des noeuds racine représentant respectivement t_1 et t_2. ⌟

Congruence. Soient $F \in \mathcal{F}(L_4)$ et `g` un e-graphe tel que `g.tset` représente $\mathcal{T}(F)$, l'ensemble des sous-termes de F. On définit la relation d'équivalence $\equiv_{\texttt{g}}$ sur $\mathcal{T}(F)$ par : $t \equiv_{\texttt{g}} t'$ si et seulement si t et t' sont représentés par des positions de `g.tset` qui sont dans la même classe d'équivalence de `g.uf`. On définit la relation $\mathcal{C}_{\texttt{g}}$ sur $\mathcal{T}(F)$ comme suit :

> ***pour tout*** $f_i \in \mathbb{F}$ ***et pour tout quadruplet de termes*** $(t_1, t_2, t_3, t_4) \in \mathcal{T}(F)^4$**,**
> ***si*** $t_1 \equiv_{\texttt{g}} t_3$ ***et*** $t_2 \equiv_{\texttt{g}} t_4$**, *alors*** $(f_i(t_1, t_2), f_i(t_3, t_4)) \in \mathcal{C}_{\texttt{g}}$**.**

Question 35. Écrire la fonction `bool term_is_congruent(egraph_t *g, int tid1, int tid2)` qui renvoie `true` si et seulement si les termes représentés aux positions `tid1` et `tid2` dans `g.tset` sont dans la relation $\mathcal{C}_{\texttt{g}}$. Donner et justifier soigneusement la complexité de `term_is_congruent` en fonction de `g.tset_sz`, si on suppose que le test d'égalité d'étiquettes (symboles de $\mathbb{X}$ ou $\mathbb{F}$) se fait en temps constant. ⌟

Algorithme de décision. Soient $F \in \mathcal{F}(L_4)$ et `g` un e-graphe tel que `g` représente les termes et les contraintes d'égalité de F ainsi que deux termes t_1 et t_2, pas nécessairement dans $\mathcal{T}(F)$. On cherche à modifier `g` pour qu'il représente $F \wedge (t_1 = t_2)$. On appelle cette opération `merge(g, t1, t2)` pour faire la différence avec `uf_union`. En effet, cette opération ne peut pas se limiter à appeler `uf_union` pour unir les classes d'équivalence des positions représentant les termes t_1 et t_2. Elle doit en plus tester si $(t'_1, t'_2) \in \mathcal{C}_{\texttt{g}}$ pour tous termes t'_1 et t'_2 qui incluent respectivement t_1 et t_2 comme sous-termes directs mais ***ne sont pas déjà dans la même***

classe d'équivalence. Si $(t'_1, t'_2) \in \mathcal{C}_{\mathtt{g}}$, alors t'_1 et t'_2 doivent être mis dans la même classe d'équivalence. Cette dernière relation est propagée aux termes qui contiennent t'_1 et t'_2 et ainsi de suite, tant qu'on trouve des couples de termes dans la relation $\mathcal{C}_{\mathtt{g}}$ mais pas dans $\equiv_{\mathtt{g}}$.

Question 36. Écrire une fonction récursive `void merge(egraph_t *g, int tid1, int tid2)` qui ajoute la contrainte d'égalité entre les termes aux positions `tid1` et `tid2` dans `g.tset`, selon le processus décrit ci-dessus. Si `tid1` et `tid2` sont déjà dans la même classe d'équivalence, `merge` termine immédiatement. ⌟

Question 37. Donner le nombre maximum d'appels récursifs qu'un appel à `merge` peut engendrer en fonction de `g.tset_sz`. ⌟

Question 38. On suppose que le graphe orienté acyclique représenté par `g.tset` contient a arêtes. Donner la complexité en temps de l'opération `merge` en fonction de `g.tset_sz` et a. ⌟

Fin du sujet.

Annexe : Preuves en déduction naturelle

Un séquent $\Gamma \vdash F$ est composé d'un ensemble de formules Γ et d'une formule F.

Le sous-ensemble suivant de règles de la déduction naturelle peut être utilisé dans la preuve demandée à la question 14.

Introduction	**Elimination**
$\dfrac{}{\Gamma, F \vdash F}\ \text{ax}$	
$\dfrac{}{\Gamma \vdash \top}\ \top_{\text{I}}$	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash F}\ \bot_{\text{E}}$
$\dfrac{\Gamma \vdash F'_1 \quad \Gamma \vdash F'_2}{\Gamma \vdash F'_1 \wedge F'_2}\ \wedge_{\text{I}}$	$\dfrac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1}\ \wedge^1_{\text{E}} \qquad \dfrac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2}\ \wedge^2_{\text{E}}$
$\dfrac{\Gamma, F' \vdash F}{\Gamma \vdash F' \to F}\ \to_{\text{I}}$	$\dfrac{\Gamma \vdash F' \to F \quad \Gamma \vdash F'}{\Gamma \vdash F}\ \to_{\text{E}}$
$\dfrac{\Gamma, F \vdash \bot}{\Gamma \vdash \neg F}\ \neg_{\text{I}}$	$\dfrac{\Gamma \vdash \neg F \quad \Gamma \vdash F}{\Gamma \vdash \bot}\ \neg_{\text{E}}$

Travailler les chapitres de ce sujet

Prepa Help suit le programme de MPI chapitre par chapitre : cours détaillé, méthodes, pièges, exercices et problèmes corrigés pas à pas, tirés des concours. L'essai gratuit donne accès aux chapitres offerts de ta filière, sans carte bancaire.

Chapitres liés à ce sujet

- Langage OCaml : expressions, types, filtrage, récursivité
- Spécifier, tester, prouver : terminaison et correction
- Récursivité
- Langage C : types, tableaux, pointeurs, allocation

Ouvrir l'essai gratuit

Scanne le code ou clique sur le bouton.

prepahelp.com/#essai

Une question sur le concours ou sur ta filière : **écrire sur WhatsApp**

Les autres années et épreuves : prepahelp.com/annales/x-ens/mpi