Projet ExceML

Modèles de Programmation et Interopérabilité des Langages

EM Alexandre

September 18, 2020

Part I
Logique et fonctionnement interne du tableur

1 Modélisation du tableur

La grille du tableur sera modélisé sous la forme d'une matrice de cases. Nous commençons donc par définir une fonction `crée_grille` : ($int \rightarrow int \rightarrow grille$), qui crée une matrice de taille $\_x \times \_y$ et l'initialise à $Vide$. Afin de pouvoir visualiser le contenu du tableur, nous avons une fonction `affiche_grille` ($grille \rightarrow unit$), qui parcourt chaque case du tableur et affiche son contenu.
Test des fonctions crées précédemment: création d'une grille, puis son initialisation sur des Cases pour l'afficher ensuite:

```
# affiche_grille (genere_grille (cree_grille 4 4));;
|@(0,0)||@(0,1)||@(0,2)||@(0,3)|
|@(1,0)||@(1,1)||@(1,2)||@(1,3)|
|@(2,0)||@(2,1)||@(2,2)||@(2,3)|
|@(3,0)||@(3,1)||@(3,2)||@(3,3)|
- : unit = ()
```

Un tableur ne doit avoir de dépendances cyclique, c'est-à-dire qu'une *expression* ne doit pas dépendre d'une autre et inversement, pour cela nous pouvons représenter la grille sous forme de graphes oriente acyclique G (DAC). On notera F la foret couvrante de G, et A ses arbres $\in F$.
Afin de vérifier qu'il n'y ait pas de cycle dans notre tableur, nous définissons alors : `cycle` ($grille \rightarrow expr$), qui parcourt chaque case de l'arbre de l'expression `expr`, seulement si celle ci n'a pas déjà été visitée.
Ce parcours est effectue par la fonction récurrente `dfs` ($grille \rightarrow int \rightarrow int \rightarrow bool$), et renvoie *true* si un noeud a déjà été visite par `dfs`, $false$ si aucun cycle n'a été détecte dans l'arbre. La fonction crée un tableau de booléens `visite` qui à chaque appel récursif de la fonction met la case de l'expression courante à `true`. On vérifie donc que chaque noeud de l'arbre est bien appelé qu'une seule fois. Test dans le cas où un des noeud (`Case (1,0)`) est dans un cycle :

```
# let has_cycle =
let gr = cree_grille 4 4 in
  gr.(0).(0) <- Entier 1 ;
  gr.(0).(1) <- Case (0,0) ;
  gr.(1).(0) <- Case (1,1) ;
  gr.(1).(1) <- Case (1,2) ;
  gr.(1).(2) <- Case (2,1) ;
  gr.(2).(1) <- Case (1,0) ;
  cycle gr (Case (1,0));;
val has_cycle : bool = true
```

Un autre test dans le cas où le noeud (`Case (0,1)`) n'est pas dans un cycle :

```
# let has_cycle =
let gr = cree_grille 4 4 in
  gr.(0).(0) <- Entier 1 ;
  gr.(0).(1) <- Case (0,0) ;
  gr.(1).(0) <- Case (1,1) ;
  gr.(1).(1) <- Case (1,2) ;
  gr.(1).(2) <- Case (2,1) ;
  gr.(2).(1) <- Case (1,0) ;
  cycle gr (Case (0,1)) ;;
val has_cycle : bool = false
```

La fonction `eval_grille` (*grille* → *resultat array array*), crée une matrice `résultat`, la parcourt de case en case, et les évalue pour entrer les résultats obtenus par l'appel de la fonction `eval_expr` (*grille* → *expr* → *resultat*).
Cette fonction évalue l'expression:

- Si c'est une `Case` alors elle fait appel à la fonction `cycle`, et retourne une `Erreur`:
  - `Cycle_détecté`, si l'expression évalué à un cycle
  - `Mauvais_indice` si les indices de la case courante sont hors de la matrice.
- Sinon elle retourne `Vide`.

Pour les autres `expressions` (`Vide, Entier, Flottant, Chaîne`), elles sont convertis en `résultat`.
Nous avons transformé les fonctions pour qu'elles implémentent un mécanisme de memoisation. La fonction `eval_grille` crée une autre matrice `mem` (*resultat array array*) initialisée à `Vide`. On la modifie ensuite dans `dfs` lors de l'évaluation de la `Case(i,j)`:

- Si `mem(i,j)` est à `Vide` alors on continue les appels récurrentes.
- Sinon on stock dans la matrice `mem` au position de la `Case` courante le résultat de `mem(i,j)`

On ajoute donc aux fonctions `dfs`, `eval_expr` et `cycle`, la matrice `mem` en argument.

La complexité de `dfs` est en $O(|V| + |E|)$, avec $|V|$ le nombre de noeuds et $|E|$ le nombre d'arêtes de l'arbre A analysé. Grâce a la memoisation, la fonction `eval_grille` n'évalue pas les noeuds de chaque arbre plusieurs fois, car on récupère le résultat obtenu précédemment. On en déduit que la complexité de `eval_grille` est en $O(|E'| \times log\ |V'|)$ avec $|V'|$ le nombre de noeuds et $|E'|$ le nombre d'arètes de G, en supposant que les opérations de memoisation soient effectués en $O(1)$, cela correspond au parcours de chaque arbre de la forêt F. Au lieu de recalculer pour chaque case de la matrice son `resultat` de l'ordre de $O(n^3)$

Pour afficher cette grille de `résultat` les fonctions `résultat_to_string` (*resultat* → *unit*) qui print le contenu de la case, si ce n'est pas un `Erreur` et lève une exception sinon. Et la fonction `affiche_grille_resultat` (*resultat array array* → *unit*)

qui parcourt chaque case de la matrice et affiche son contenue grâce a l'appel de `résultat_to_string`.

Test des fonctions avec la grille `gr` crées précédemment :

```
# let res = eval_grille gr;;
val res : resultat array array =
  [|[|Entier 1; Entier 1; Vide; Vide|];
    [|Erreur Cycle_detecte; Erreur Cycle_detecte;
      Erreur Cycle_detecte; Vide|];
    [|Vide; Erreur Cycle_detecte; Vide; Vide|];
    [|Vide; Vide; Vide; Vide|]|]

# affiche_grille_resultat res;;
|    1||    1||     ||     |
Exception: Has_cycle "Cycle_detecte".
```

2 Formules

Pour les nouvelles expressions : **Unaire, Binaire, Réduction**, nous complétons les fonctions `dfs` et `eval_expr`. Nous faisons appliquer les fonctions passes dans la structure de donnée (*app1,app2*), aux opérateurs (**Unaire, Binaire**) respectivement. Et pour **Réduction**, selon les cases de début et fin nous évaluons l'ensemble de cases de cases entre ces deux cases, de droite a gauche ou inversement grace aux fonctions `op_red_left` ou `op_red_right`. Lors de l'affichage de la grille, la fonction `affiche_grille` fait afficher les opérations **Unaire, Binaire, Réduction** respectivement sous la forme : `opUn, op2, opRed`.

La fonction `abs` prend une expression `n`, soit un `Entier`, soit un `Flottant` et renvoie une expression `Unaire(app1=f;opérande)`, avec `abs_expr` une fonction qui prends un résultat n et renvoie sa valeur absolue en `résultat`. Lève une exception lorsque le type n'est pas un `Entier/Flottant`

La fonction `add` prends deux expressions `g,d`, et génère une expression sous la forme `Binaire(app2=addres;gauche=g;droite=d)`, `addres` étant une fonction qui génère la somme de deux `résultat`. g et d sont des expressions (Entier,Flottant, Chaîne).

La fonction `somme` prend une case de début et une de fin en paramètre, et génère une expression sous la forme `Réduction(app=h;init=ini;case_début;case_fin)`, `ini` correspond à la valeur initiale.

```
# let gr = cree_grille 4 4 in
  gr.(0).(1) <- (Chaine("je"));
  gr.(0).(3) <- (Chaine("suis"));
  gr.(1).(2) <- (Chaine("la"));
  gr.(0).(0) <- (Case(1,2));
  gr.(3).(3) <- somme (1,2) (0,0);
# let res = eval_grille gr;;
```

```
# affiche_grille_resultat res;;
|   la||    je||      || suis|
|     ||     ||   la||      |
|     ||     ||     ||      |
|     ||     ||     ||lasuisjela|
```

De la même façon, on implémente les opérations unaires : **Inverse, Opposé**, binaires : **Multiplication, Division**, réduction : **Min, Max**
Test avec mélanges d'expressions

```
# let gr = cree_grille 4 4;;
  gr.(0).(0) <- Entier 2;
  gr.(0).(1) <- Entier (-99);
  gr.(0).(2) <- Entier 99;
  gr.(2).(0) <- Entier 10;
  gr.(2).(1) <- Entier 2;
  gr.(1).(0) <- Entier 5;
  gr.(3).(0) <- div (Case(2,0)) (Case(2,1));
  gr.(1).(1) <- oppose (Entier 5);
  gr.(3).(1) <- mul (Case(1,0)) (Case(1,1));;
  gr.(3).(2) <- minred (2,3) (0,0);
  gr.(3).(3) <- maxred (0,0) (2,3);


# let res = eval_grille gr;;
val res : resultat array array =
    [|[|Entier 2; Entier -99; Entier 99; Vide|];
      [|Entier 5; Entier -5; Vide; Vide|];
      [|Flottant 10.; Flottant 2.; Vide; Vide|];
      [|Flottant 5.; Entier -25; Entier -99; Entier 99|]|]

# affiche_grille_resultat res;;
|    2||  -99||   99||     |
|    5||   -5||     ||     |
|  10.||   2.||     ||     |
|   5.||  -25||  -99||   99|

- : unit = ()
```

Part II
Interface graphique

2 Charger les cellules

Pour charger les cellules du tableur, on définit la fonction :

$$\texttt{build\_cell} : (infos\_grid \rightarrow cell\_infos) \tag{1}$$

, où `infos_grid` est un `cell_infos Array Array`.
On utilise tout d'abord la fonction `mk_cell`:$(unit \rightarrow cell\_info)$, pour la création d'une cellule vide, on utilise ensuite la fonction `Dom.appendChild` afin d'ajouter la cellule donc `container` dans le tableau `cells`, nous faisons la même chose avec `txt` et `inp` ajoute au `container` et renvoyer cellule crée.
Pour afficher les cellules dans le navigateur, nous ajoutons à chaque cellule la classe CSS `cell-container` grâce à la fonction `Dom.Class.add` qui permet de mettre à jour le contenu de `container` de la cellule courante.

3 Gestion des évènements

On définit la fonction :

$$\texttt{add\_cell\_events} : (int \rightarrow int \rightarrow grid \rightarrow infos\_grid \rightarrow unit) \tag{2}$$

qui permettra de gérer les interactions avec les cellules.

- Lorsque l'utilisateur fait un **double-clique** sur une case du tableur, on fait appel à la fonction `Dom.Events.set_ondbclick` et on lui applique une fonction handler $(unit \rightarrow unit)$:

```
let f () = Dom.Class.add cell.inp "editing-input";
           Dom.Focus.focus cell.inp
```

  On active ainsi le mode édition en ajoutant la classe `editing-input` à l'input de la cellule et on active le focus sur la cellule courante.

- Lorsque l'utilisateur tape au clavier pour inserer une valeur dans le tableur, on fait appel at la fonction `Dom.Events.set_onkeydown` et on lui applique une fonction handler $(int \rightarrow boolean)$:

```
let h (v:int) =
    if v=13 then
        (update i j grid infos_grid;
        Dom.Class.remove cell.inp "editing-input");
    true
```

  qui prends le code ASCII de la touche tapé. Et lorsque l'utilisateur tape sur la touche *Enter*, on quitte l'édition de la cellule, on sauvegarde la valeur entré par l'utilisateur dans `cell.txt`, on a ensuite implémenté une fonction `update` qui sera décrite dans la partie **5.1**.

- Lorsque l'utilisateur quitte le mode édition, on sauvegarde la valeur entré et on quitte le mode focus avec la fonction handler ($unit \to unit$)

```
let g () = Dom.Focus.blur cell.inp;
           update i j grid infos_grid;
           Dom.Class.remove cell.inp "editing-input"
```

4 Stockage des données

Pour stocker les valeurs inseres dans le tableur, on utilise les fonctions `Stockage.set` et `Stockage.find`.

On aura donc besoin de transformer le tableur en chaîne de caractères, on définit donc la fonction

$$\texttt{grid\_to\_string} : (grid \to infos\_grid) \tag{3}$$

, qui parcourt le tableur et on stock les valeurs d'input et des résultats déjà évalués sous le format `i|j|value` où i et j sont les position de la cellule dans le tableur et value la valeur à stocker. Chaque valeur est séparé avec le caractère '\n'. La condition a la fin de la fonction permet d'enlever le dernier '\n' et d'éviter d'ajouter un élément à la liste cree dans :

$$\texttt{cell\_of\_string} : (string \to (int * int * string)\ list) \tag{4}$$

Elle permet de récupérer la liste de cellule sauvegardé, en séparant la chaîne de caractères a chaque '\n' grâce à la fonction `String.split_on_char`. On parcourt ensuite la liste et on fait la même chose avec le caractère $'|'$ et on ajoute $(i, j, value)$ à la liste résultat en matchant et en les transformant en $(int, int, string)$.

Au chargement de la page, on fait appel à la fonction :

$$\texttt{load\_storage} : (grid \to infos\_grid \to unit) \tag{5}$$

, qui charge la chaîne de caractère sauvegarder par la fonction `Stockage.set`, puis on la transforme avec la fonction `cells_of_string`. On parcours ensuite la liste cree par la fonction précédente et en matchant chaque élément, pour `value` qui est un string, on appelle la fonction `Ast.make` qui permet de la transformer en `Tableur.expr`. On stockera l'expression obtenu dans `grid` et en mettant à jour `infos_grid` au position (i, j).

5 Évaluation d'une case

5.1 Du texte vers l'expression

Lorsque l'utilisateur entre une expression dans une cellule, on récupère l'expression sous la forme de chaîne de caractère, on appelle donc la fonction `Ast.make`. Il faudra alors matcher avec `Ok(expr)` et stocker `expr` dans `grid` et si input est non

évaluable alors, il est matché avec `Error(e)` et on affichera "ERROR" sur le tableur.

On fera appel à `update` dans `load_storage` à chaque changement de valeur d'une cellule.

5.2 De l'expression au résultat

L'expression stocké sera ensuite évalué avec la fonction de la partie 1 du projet `Tableur.eval` si l'utilisateur n'entre pas une division par zero, on attrapera l'exception et on la transformera en `Erreur`.
Le résultat sera ensuite stocké dans `infos_grid.(i).(j).result`.

5.3 Mise à jour graphique

La mise à jour graphique se fera dans la fonction:

$$\texttt{update\_display} : (infos\_grid \to int \to int \to resultat \to unit) \tag{6}$$

Pour afficher le résultat obtenu, on définit la fonction `resultat_to_string` qui transforme le résultat en chaîne de caractère que l'on insérera dans `inp` et `txt` de la cellule courante dans la fonction `update_display`. Si le pattern matching match le résultat avec une erreur, on transformera l'erreur en chaîne de caractère grâce à la fonction `error_to_string` et on ajoutera à la cellule la classe `cell-error`.

6 Propagation de la mise à jour

Pour mettre à jour les dépendances d'une cellule, on définit la fonction :

$$\texttt{update\_deps} : (infos\_grid \to int \to int \to expr \to unit) \tag{7}$$

qui met à jour les listes de dépendances $((int * int)list)$ `parent_deps` et `child_deps`. Avec la fonction `direct_deps`:$(expr \to (int * int)list)$, on génère la liste des cellules dont elle dépend directement puis on parcourt cette liste pour mettre a jour les listes de dépendances `child_deps` des cellules de la liste.

$$\texttt{propagate} : (grid \to infos\_grid \to int \to int \to unit) \tag{8}$$

On veut ensuite propager les résultats et l'affichage des cellules enfants en parcourant la liste `cell.child_deps` en appelant la fonction `update_display`. Elle sera utilisé à la fin de la fonction `update`.

*Pour effacer le contenu d'une cellule, il faut effacer son contenu (**`inp`**) puis entrer un espace vide c'est à dire " ".*