TP2 IFT 2035

Tchoumkeu Djeumen Thierry Manuel 20170651
Medjahed Mehdi 20142385

13 Décembre 2021

RAPPORT

1 Introduction

Voici donc le rapport de notre TP numéro 2 du cours IFT 2035 Automne 2021.

2 Élaboration

2.1 Expression booléennes

Ce sont respectivement "true" et "false". L'élaboration de ces derniers n'a pas été compliquée car comme pour les entiers, on doit juste retourner la meme expréssion avec le type "bool" au lieu de "int" pour comme pour les entiers et c'est ce qu'on a fait.

2.2 L'inférence des types

Au début nous avons décomposé l'expréssion en 3 éléments, Head, Middle et Tail pour ensuite élaborer le Middle. Nous avons remarqué qu'à la compilation, cela a généré un warnirng d'instanciation d'un "Singleton", en d'autres termes, on disait que le Head n'était pas utilisé. Nous avons donc retirer la variable Head et l'avons remplacé par une "_" pour que le compilateur ne prenne pas en compte le premier élément (":"). Lorsque nous avons voulu tester ce code avec l'exemple "nil : list(bool)", nous avons fait face à une erreur d'instanciation. Lors de l'exécution avec la trace, nous avons remarqué que le compilateur reconnaissait l'expression comme une expression arithmétique car leurs décompositions avec l'opérateur "=.." renvoyait un tableau avec le meme nombre d'éléments. Pour corriger cela, nous avons implémenter un cas spécial de décomposition. Au lieu de mettre "[Head, Middle, Tail]", on a mis "[: Middle, Tail]" et nous avons positionné ce fragment de code avant l'élaboration des expressions arithmétiques, pour que cela soit un cas differenciable de celui des opérations arithmétiques, et qu'il soit vu avant par le compilateur.

2.3 Inverse de l'inférence des types

Il fallait tout d'abord, tenir compte comme pour l'inférence de types du premier élément issu de la décomposition de l'expression E, nous la reconnaissons que lorsque la décomposition donne en premier élément le "?". L'expression à élaborer est le second élément de l'expression décomposé. Le troisième élément du tableau correspond au type que l'on va envoyer. Nous avons un cas de base, pour une expression simple de type "?(1,nil), notre implémentation ne prend en charge que les cons, et nous avons remarqué que l'élaboration était exactement la même que celle des cons, nous avons donc fait en sorte de faire appelle à l'elaborate des cons pour élaborer les expressions avec "?". Les problèmes que nous avons eu au début étaient liés au fait que nous ne comprenions à partir de l'exemple donné sur le forum de StudiUM comme traiter une expression du même type que ?(1,nil), ce n'est qu'après que nous avions compris qu'il fallait les traiter comme des listes.

2.4 Expressions Let

Cela fut l'un des plus compliqués à faire car il nous a fallu mettre à jour les variables instanciées dans l'environement de base et faire une élaboration de l'expréssion de l'opérateur let en question. Pour ce faire, nous avons donc appelé réccursivement le prédicat elaborate mais cette fois si avec la variable nouvellement instanciée et son type en premier élément de l'environement(In dice de DeBruijin). L'expréssion retournée est donc une expréssion let avec chacune de ses composantes déhà élaborées pour l'évaluation.

2.5 Constructeur de listes (cons)

Nous avons remarquer la grande ressemblance entre l'élaboration des expression arithmétiques et celles des listes qui fonctionnent de la même manière, à la différence que le dernier élément de la liste est toujours un nil (liste vide), ce qui nous permet de les reconnaitre et les differencier des opérations arithmétiques. Donc, on a créé un cas de base qui élabore cette liste vide, et qui est très important pour arréter la récursion. Pour le cas général : en ce qui concerne le premier élément du cons, on a utilisé l'implémentation triviale des chiffres qui retourne un int. En ce qui concerne l'élément du milieu de la liste générée par '=..', celui-ci est élaboré réccursivement de la même façon que pour les opérations arithmétiques. Pour les types, cela rend un type générique qui est inscrit dans l'environnemment de base, par faute de temps nous n'avons pas pu arranger ça. Notre idée pour arranger cela, aurait été de vérifier le premier élément de la liste et de retourner list(le type du premier élément).

2.6 Opérateurs sur les listes: empty, car et cdr

Nous avons décomposer notre Expression avec l'opérateur "=.." et chacun est décomposé en une liste avec comme premier élément soit empty, car, ou cdr,

et nous avons fait l'appel correspondant à chacun de ses opérateurs en sortie. Chacun des opérateurs sera pris dans l'environement et élaborer en utilisant les indices de De Bruijin via le prédicat "index" que l'on a implémenter plus bas dans le code.

2.7 Les opérations arithmétiques

Nous nous sommes basés sur les exemples de code à élaborer donnés sur le forum Studium, nous avos chercher à decomposer ce code en une liste de plusieurs éléments, avec le premier élément qui sera l'opérateur qu'on va chercher dans l'environement, puis l'on a élaborer le reste. Pour décomposer la fonction, on a penser à implémenter la décomposition via un avec un nouveau prédicat auxilliaire, car on n'avait pas encore compris le fonctionemment de l'opérateur prolog '=..'. On a eu aussi du mal avec l'annotation infixe ou préfixe des expréssions qui allaient etre données. Puis avec disscussion avec le professeur, on a pu comprendre et on a utiliser l'opérateur '=..' pour segmenter notre expression, et il s'est révélé etre un outil très puissant et très utile, qui nous a beaucoup aidé, car on l'a utilisé tout au long des implémentations des prédicats elaborate. On a rejeté l'option du prédicat pour la décomposition car il représentait un travail supplémentaire long et inutile. Tout d'abord, on a décomposé les prédicats en un cas de base qui prend une expression avec un + et un nombre, que nous avons décomposé avec "=.." en Head et Eretour. Pour obtenir le type de l'expression on a créé une fonction index, qui fait pratiquement la même chose que nth-elem et retourne le tuple issu de l'envirronnement qui contient l'opérateur et le type, qui est retourné par la suite. Pour le cas récursif, le "=.." Décompose en trois parties, le +, et deux autres expressions

2.8 Opérateur conditionnel

On a décomposé l'expression a elaborer avec l'operateur "=.." et on à vérifier si la tete du tableau était bien un if. Les trois expressions suivantes correspondents la condition (E1), puis l'expression then (E2) et l'expression else (E3). Cette élaboration à été la plus simple et la plus intuitive à faire, car nous avions a faire seulement l'élaboration de chacune des expressions, et renvoyer en expressions finales, seulement le if avec les expressions élaborées.

2.9 Lambda expression

Nous avons utiliser l'opérateur "=.." pour subdiviser la lambda expression et vérifier si la variable de l'expression était bien un identifiant, avant de retourner la lambda expression élaborée. La difficulté que pour la plupart des cas ici est de ressortir le type de l'expression. Nous avons remarqué que ce que nous avons fait, fonctionnait dèja avec l'implémentation du devoir. Donc Malheureusement, par fautes de temps nous n'avons pas pu terminer l'affichage des types, cela à été notre principal problème dans ce devoir. Il aurait fallu, pour les types

polymorphes, les généraliser avec la règle donné "generalize" pour générer un type polymorphe.

3 Evaluateur

L'evaluation via le prédicat "runeval" va tenir compte de l'environement contenant uniquement les "builtin". Après avoir fait ce constat, nous avons adapté notre implémentation car à la base, nous pensions que l'environement qui devait etre considéré serait l'environement initial(env0).

3.1 Constructeurs de listes

Pour les constructeurs de listes, nous avons remarqué que dans le code du devoir il y'avait dèja une fonction permettant de traiter récursivement les app, nous avons donc simplement ajouté le cas de base, qui est reconnaissable à travers l'expression élaborée de cons (app(app(var(Idx), A), var(Indx2)), nous avons utilisé la régle builtin pour chercher le builtin de cons dans l'environnement grace a l'index Idx, et le premier appel construit la premiere partie de la liste, puis pour la deuxième partie on appelle le builtin de X, avec la liste vide, ce qui correspond à la fin de la liste. Par ailleurs nous avons tout d'abord commencé par implémenter les operations arithmetiques puis nous avons fait celle des constructeurs de listes, et cela à engendré un problème à leur évaluation, car lorsqu'on voulait évaluer une liste le compilateur additionnait tous les éléments de la list, nous avons ainsi remarqué que la forme d'élaboration des listes et celle des opérations arithmétiques était très similaire, la facon de les differencier est au niveau de builtin(X, [],V) qui s^p*écifiequ'uneliste finitavecdernierélémentunelistevide.Pourreglerleproblè*

3.2 Opérateurs de liste car, cdr et empty

Nous avons donc utilisé l'environement entré en argument du prédicat (celui contenant les builtin) pour pouvoir implémenter ces opérateurs. Pour ce faire, nous avons donc fait un cas de base d'utilisation de ces opérateurs là dont nous allons chercher leur builtin dans l'environement grace à leur indice de DeBuijin. On a ensuite évaluer l'expréssion qui était venus en entrée avec l'opérateur de liste pour avoir une liste au bon format "[]", puis nous avons employé le prédicat builtin en fonction de l'opérateur pris en entrée et nous l'avons appliqué sur la liste fraichement élaborée pour pouvoir avooir le résultat final.

3.3 Expressions arithmétiques

Nous avons réussi au début à implémenter seulement le cas de base + (quelque chose) qui ne renvoie pas un résultat, puis dans le cas récursif rend possible l'addition de deux nombres ou plus, l'opération d'addition se fait grâce au builtin qui retourne le résultat de l'opération. Dans eval(Env, app(var(Indx), A), V) par exemple, nous avons Indx qui correspond à l'indice ou se trouve le + dans

l'environnement, on applique la règle builtin, à l'élément qui est positionné à l'index grâce à la fonction index que nous avons implémenter.

3.4 Let

Pour l'évaluation des expréssions "let", nous avons tout d'abord ajouter à l'environement de l'appel de l'évaluation les résultats des évaluations des variables nouvellement instanciées grace au let. Puis, nous avons fait en sorte que, dans l'expréssion dont on doit renvoyer l'évaluation, chaque instace de variable (x ou y par exemple) que l'on retrouve soit remplacé par sa valeur contenu dans l'environment. Les variables ici sont représentées grace au indices de DeBruijin et on se sert de ça pour les retrouver dans l'environement utilisé pour l'appel du prédicat eval.

3.5 L'opérateur conditionnel "if"

En ce qui concerne cet opérateur, nous avos juste évaluer de façon réccursive la condition et les expressions retournées en fonction de si la condition est vrai ou pas. Si la condition est vraie, la valeur de retour sera l'évaluation de la première expréssion E1; sinon, ce sera celle de E2. Par contre, nous avons eu un soucis au niveau de la syntaxe des conditions en prolog qui nous permettaient de faire cela donc, l'évaluation de cet opérateur conditionnel n'est pas très optimal et ne fonctionne vraiment que dans un seul des deux cas, soit si la condition est true ou false en focntion de la position dans le code.

4 Conclusion

Nous avons donc pu faire la majorité des cas d'évaluation et d'élaboration requis pour ce devoir. Malheureusement, les types de retour pour chaque expréssions ne sont pas forcément corrects pour toutes les expréssions et l'évaluation de certaines expréssions assez complexes ne sont pas optimal. Faute de temps et de compréhension, nous n'avons pas pu bien faire cela.