Université de Montréal

Rapport : Langage OUF!

Par  
Laura Bégin  
Raphaël Guillemin

Baccalauréat en informatique  
Faculté des Arts et des Sciences

Travail présenté à Vincent Archambault-Bouffard  
dans le cadre du cours IGY-2035  
Concepts des langages de programmation

24 juin 20192

**Structure du programme**

Après avoir regardé rapidement les tests unitaires, nous avons déterminé que nous allions suivre l’ordre des tests pour construire l’évaluateur *Ouf*! La première étape était donc les entrées *define* utilisées pour stocker des variables et des fonctions dans l’environnement. Pour la traduction du langage *Ouf*! vers une forme évaluable, nous avons ajouté trois cas dans la fonction *specialForm2Exp*. Si *define* n’est pas suivi d’argument, la fonction renvoie une erreur. Si la fonction reçoit un nom de variable et un nombre, on transforme la variable en symbole et le nombre en expression évaluable. Enfin, si la fonction reçoit en paramètre une autre fonction, tel qu’une fonction lambda par exemple, un appel récursif se passe jusqu’à ce que la dernière fonction soit traduite pour l’évaluation. Pour la partie d’évaluation des fonctions *define*, nous utilisons tout simplement la fonction *insertVar* pour ajouter la variable à l’environnement.

La deuxième étape de l’évaluateur consiste à lire des entrées de primitives (+, -, \*) en préfixe et de les évaluer. Pour la traduction, nous utilisons la fonction *sexp2Exp* qui vérifie que le nombre d’arguments est suffisant et transforme l’équation en *EApp* en fonction du nombre et du type d’arguments passés. Pour l’évaluation des *EApp*, nous séparons l’évaluation en deux cas : un pour les primitives et l’autre pour les fonctions lambda.

Ensuite, pour les expressions lambda, nous effectuons la traduction en *ELam* grâce à un appel récursif de *specialForm2Exp* implanté dans le cas du *define* afin de passer au cas du *lambda*. Pour l’évaluation, nous transformons le corps de la fonction lambda en *VLam* et nous retournons sa valeur et son environnement.

Quant aux expressions *let*, nous appliquons une fonction anonyme dans un *map* sur toutes les variables passées en argument, ce qui nous permet de séparer les éléments importants pour les transformer en *Symbol* et en *Exp* qui seront utilisés pour l’évaluation. On construit donc le *ELet* avec la liste de *Symbol* et d’*Exp* avec la dernière expression du *let*, qui est la fonction qu’on cherche à faire. Pour l’évaluation, nous séparons encore une fois les éléments du *ELet* avec une fonction lambda dans un *map* afin de faire des nouveaux tuples de *Symbol* et d’*Exp*. Nous mettons le tout dans un environnement fermé afin de ne pas pouvoir accéder aux variables depuis l’extérieur du let.

La fermeture fonctionne comme prévu pour les *let*, les variables sont mémorisées au sein de la fonction mais ne sont ni accessible par les autres fonctions, ni accessible au niveau global.

Pour la traduction des *data*, nous séparons les arguments en 3 parties, le nom du type de *data*, et les deux constructeurs. Nous avons ajouté une fonction *dataConst*, qui nous permet de renvoyer des *DataConstructor* à partir d’une *Sexp*. Nous utilisons cette fonction sur les deux constructeurs et nous les concaténons pour les renvoyer avec le nom du type de *data* dans un *EData* pour l’évaluation. Pour l’évaluation d’un *data*,

**Expérience de développement**

***Difficultés***

Lors de la conception de l’évaluateur *ouf!*, nous avons été confrontés à une multitude de problèmes. Chaque étape des tests unitaires était parsemée d’erreurs de tous types. Tout d’abord, en commençant le projet nous ne connaissions pas bien Haskell et nous avons tenté de comprendre le code. Nous avons finalement compris le fonctionnement de la fonction *sequence* qui est essentielle pour compléter ce projet. Le premier petit problème que nous avons rencontré était une petite erreur de nom de variable de retour d’environnement, qui faisait en sorte que lorsque nous définissions une variable, elle ne restait pas dans l’environnement global.

Par la suite, nous avons eu du mal lors de la traduction des primitives. Il fallait séparer les arguments de différentes manières pour que la fonction soit la plus globale possible. Pour l’évaluation nous n’avions pas compris que *primDef* était dans l’environnement et qu’en utilisant *EVar* dans l’évaluateur *EApp*, l’évaluateur utilisait *lookUpVar* pour trouver les variables dans l’environnement pour l’évaluation.

Ensuite, pour les expressions lambda, nous avons eu beaucoup de mal à les intégrer à l’environnement pour pouvoir les utiliser ensuite. Nous avons également eu un problème pour mettre plusieurs arguments dans une fonction lambda et de retourner la bonne valeur lors de l’évaluation. Mais encore, nous avons eu de la difficulté à comprendre comment utiliser notre fonction lambda avec toutes les parenthèses comme dans le test unitaire (((bar 4)5)6). Nous avons finalement dû faire un cas à part pour quand les arguments sont tous déjà séparés par des parenthèses.

Nous avons aussi éprouvé de la difficulté à traduire le *let* car nous n’arrivions pas à utiliser la fonction *sequence* avec une fonction lambda imbriquée. Nous avons passé beaucoup de temps à comprendre la fonction, mais une fois que nous l’avons comprise, tout est devenu plus simple à coder. Pareil pour l’évaluation du let, nous avons eu de la difficulté à séparer les arguments comme il fallait jusqu’à ce que l’on utilise la fonction *sequence*. Une autre partie difficile de la fonction *let* était de garder les variables dans un environnement à part et de les utiliser dans les fonctions écrites dans le dernier paramètre du let.

Enfin, l’endroit où nous avons eu le plus de difficulté est lors de l’implémentation de *data.* Il a fallu trouver comment séparer chaque partie de l’expression data pour les traduire dans des expressions que le programme peut évaluer. Nous avons ajouté une fonction *dataConst* qui retourne un *DataConstructor*, ce qui est nécessaire pour l’évaluation. Lors de l’évaluation nous avons dû déterminer que nous devions utiliser VPrim dans VData, ce qui a pris une journée complète. Avant d’avoir compris cela, nous avions créé une autre fonction, mais elle évaluait seulement les constructeurs ne prenant aucun argument comme *True*. Il s’agissait simplement de retourner *VData type const [VUnit]*. Après plusieurs essais infructueux, nous avons enfin réussi à analyser les *data* prenant un seul argument comme *Just Int*. Nous avons hésité longuement quant au type *Int* : est-ce que le langage le connaissait déjà? devions nous l’ajouter à l’environnement? Notre décision s’est arrêtée à un filtrage par motif du mot *Int* qui renvoie un *VPrim (\(Vint x) –> (Vint x))*.

***Fonctions manquantes***

Nous n’avons pas eu le temps de compléter les fonctions *case* et la portée dynamique.

Pour la fonction *case*, notre algorithme ressemblerait à :

Pour la portée dynamique, nous ferions :

***Globale***

Dans l’ensemble, nous sommes satisfaits du résultat obtenu. Nous avons passé beaucoup de temps à essayer de comprendre l’ensemble des fonctions du programme et nous avons fini par résoudre la plupart de nos problèmes. Nous n’avons pas eu le temps de terminer tout le travail car nous avons passé beaucoup de temps sur la fonction *data.* Nous avons décidé de mettre toute notre énergie à évaluer les *data* puisque les consignes du travail conseillaient d’implanter les fonctionnalités dans l’ordre des tests unitaires. Comme la partie suivante semblait encore plus ardue (portée statique), nous avons pris la décision de réussir *data* avant tout.