Rapport projet 2, Fouine

Dziki Yanis et Gardies Vincent

1 Présentation

Voici notre fouine, codée en Ocaml, et traitant la majorité des fonctionnalitées attendues à l'exception du polymorphisme.
Les instructions de compilations sont classiques, il suffit de make et d'éxécuter `./fouine` suivi des options décrites ci-dessous.

2 Organisation du code

Le code est structuré de la manière suivante :

— Un fichier `main.ml` permettant la gestion des options et l'éxécution du code adéquat

— Un fichier `expr.ml` se focalisant sur évalutation du code passé en entrée à la fouine

— Un fichier `affichage.ml` servant à formater l'affichage de la sortie de la fouine

— Un fichier `options.ml` contenant les références associées aux options disponibles

— Un fichier `inference.ml` permettant d'effectuer l'inférence de types

— Un fichier `unif.ml` exécutant l'algorithme d'unification sur le problème de sorties de l'inférence de types.

— Deux fichiers `lexer.mll` et `parser.mll` effecutant respectivement la tâche de lexer et parser.

3 Descriptif des options

Les options disponibles par notre fouine sont :

— L'option `-trace` qui permet de suivre les états de l'automate durant le parsing de l'expression entrée.

— L'option `-warnings` qui propose les warnings présents dans Ocaml (par exemple le type de la première expression dans une séquence devrait être unit).

— L'option `-output` qui permet de renvoyer la sortie de l'expression entrée.

— L'option `-showsrc` qui permet d'afficher le code Caml correspondant à l'expression fouine.

— L'option `-debug` qui permet d'afficher les états de l'environnement durant l'évaluation et l'inférence.
— Une option `-slow` qui correspond à un debug étape par étape de l'évaluation.

— L'option `-showtypes` qui permet d'afficher les types de l'ensemble des variables crées dans l'expression (variables globales, locales et auxiliaires) de la fouine

— L'option `-notypes` qui effectue le comportement de la fouine sans inférence de types

— L'option `-showinf` est un debug qui permet de suivre les étapes de l'inférence de types.

— L'option `-tree` qui affiche l'arbre d'instructions correspondant à l'expression entrée.

4 Tests effectués

Nous avons testé notre fouine sur la moulinette de tests disponibles à laquelle nous avons rajouté des tests afin de vérifier que le typage fonctionnait, et que la sortie était une sortie conforme à ce que Ocaml donnait en sortie, cela nous a donc incité à afficher les sorties dans un certain ordre et à renommer les variables de types du monomorphisme.

5 Améliorations possibles

— Nous nous sommes rendus compte que notre compréhension de la manière d'adapter l'algorithme d'unification pour l'inférence de type était différente de ce qui était conseillé dans les notes de cours trop tard, et nous avons donc décider de continuer sur notre modèle qui était, a posteriori plus

compliqué que ce qui était conseillé et nous avons donc perdu du temps que nous aurions pu utiliser pour l'implémentation du polymorphisme.

6 Points Notables

— Nous avons traité les expressions `Match .. With` en nous basant sur notre fonction filtre qui nous permet de filtrer les expressions qui correspondent à notre motif. Nous avons donc créé une fonction qui prend en paramètre une expression , un environnement et un motif et qui renvoie un booléen et un nouvel environnement, le booléen permettant de savoir si un cas de matching correspondait. Nous avons ensuite utilisé cette fonction pour traiter les expressions Match .. With.

— Nous avons procédé de manière presque similaire pour `Try .. With` mais en rajoutant des motifs correspondants aux expressions présentes dans le With et correspondant aux exceptions possibles.

— Les références, les listes et les séquences d'expression sont implémentés en suivant les indications données dans les notes de cours

— Les fonctions récursives diffèrent des fonctions non récursives grâce à un booléen qui est présent lors de la déclaration de la fonction et qui est mis à true si le mot clé `rec` est présent. Ainsi lorsque la fouine est sur le point d'appliquer une fonction qu'elle sait récursive, elle va renvoyer la valeur de la fonction (qui correspond à un motif, un corps, un booléen pour la récursivité et un environnement) en se rajoutant elle même à son propre environnement. De cette façon toute fonction récursive à accès à elle-même dans son environnement.

— Comme expliqué précédemment, nous avons dérivé sur notre manière de procéder au typage des expressions, premièrement en modifiant le type `t` de l'algorithme d'unification permettant de décomposer les termes en remplacant l'opérateur `Op` destiné à récupérer les arguments des fonctions par un traitement "récursif" des fonctions similaires au traitement du parser pour ces fonctions. De cette façon, on conserve l'esprit de Caml dans le sens où toute fonction possède un unique argument. Notre type `t` est récursif et composé de `Var` correspondant aux variables (aussi bien globales que locales ou auxiliaires), de `T` correspondant aux types fondamentaux (ex : int, bool, ref, fun, ...), de `Prime` correspondant aux T0/weak/'a et enfin de `None` qui correspond à l'absence d'information concernant un type. Deuxiemement, notre inférence a pour but de générer un ensemble de contraintes de typages que l'on stocke dans une liste ref et s'effectue par un parcours de l'expression couplé à la fonction `find_type : expr -> t` dont le but est de déterminer le type d'une expression, ces deux fonctions vont parcourir "parallèlement" la totalité de l'expression. Ce

double parcours nous a posé un problème pour les cas où certains noms de variables étaient utilisés plusieurs fois dans des contextes différents. Pour pallier à ce problème, nous avons opté pour un renommage des variables avant de procéder à l'inférence de type en ajoutant des entiers devant chaque variable de telle sorte que chaque variable possède un nom unique, on redirige alors la nouvelle expression obtenue vers les algorithmes d'inférence de type et d'unification.

— Concernant l'unification, on parcours la liste des inférences en boucle en supprimant les couples qui ne posent pas de problèmes (ex : int, int ou None, bool), en décomposant les types imbriqués (ex : fun, tuples, list) et en remplacant les variables ou prime par leur type lorsqu'ils sont fixés jusqu'a ce que la liste soit vide. Nous avons décidé d'implémenter une fonction `type_fixed : t -> bool` qui permet de déterminer si un type est établi lors de l'unification (càd qu'il ne dépend pas d'une variable, ou d'un type 'a inconnu), le traitement des variables de type générique (T0/T1... que nous avons appelé `Prime`) y est alors spécifique puisque on s'assure que l'algorithme d'unification ait déjà observer toutes les contraintes avant de fixer un tel type.

7 Passage obligé

Voici une référence [1] de Peter Landin

Références

[1] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3) :157–166, 1966.