TD2 : interpréteur du langage du cours

2023-2024

Le sujet consiste à implémenter un interpréteur pour le langage décrit en cours sur la machine abstraite également décrite en cours.

Exercice 1: Travailler aussi sur papier

Pour bien comprendre ce que doit faire l'interpréteur, la notion de sémantique, ainsi que s'entraîner à ce qu'on vous demandera de savoir faire le jour de l'examen, on va aussi dérouler des exemples sur papier dans des mini-langages. Faites donc, en parallèle de l'exercice suivant (qui est du code) les exercices sur la feuille d'exercices papier accompagnant ce sujet (trouvable sur la page du cours).

Sur cette feuille, les exercices concernent aussi les sous-langages considérés pour notre langage du cours, aussi, faites-les au rythme où vous en êtes dans l'implémentation.

Exercice 2: Langage du cours Le langage et sa sémantique sont décrits dans le chapitre 2 du polycopié, on ne le recopiera donc pas ici. Globalement, le polycopié a été écrit (modulo des éventuelles typos) de manière à ce que la syntaxe corresponde à celle du type utilisé pour représenter les expressions, instructions et déclarations de fonctions. Cela dit, en cas de conflit, c'est évidemment la version présente dans le code qui prime.

En particulier, il y a une différence sur les `Return` : dans le poly, par simplicité, on a deux instructions (une avec un argument, l'autre sans), quand ici on n'en a qu'une seule renvoyant un type option (`None` pour pas d'argument, `Some v` pour un argument).

Le projet est organisé, à l'intérieur du dossier `compiler` comme suit :

- Un dossier `bin` qui contient le code de l'exécutable. Vous pouvez l'ignorer.
- Un dossier `examples` qui contient des définitions d'expressions, d'instructions et de programmes que vous pourrez tester. Vous pouvez éventuellement y regarder des définitions de certains programmes, mais la syntaxe est complexe, il sera plus facile de faire afficher les exemples à l'exécutable. Vous pouvez ajouter vos propres exemples dans le fichier `examples_generated.ml` (dans les listes existantes). Ceci dit, les programmes présents dans ce dernier fichiers sont générés à partir des programmes écrit dans un langage plus naturels situés dans le dossier `programs` du td (uniquement les programmes corrects). Vous pouvez également regarder ce format-là (mais malheureusement pas le modifier pour le moment).
- Un dossier `util` qui contient des modules utilitaires pour l'ensemble du compilateur. Pour le moment, il contient uniquement le module `Environment` qui vous servira pour représenter la mémoire de la machine abstraite.
- Un dossier `three_address` qui est vide pour le moment (il servira à la fin du cours, mais pour des raisons de compatibilité de la doc, il doit être présent).

- Un dossier `abstract_machine` qui contient la définition de la machine abstraite, que vous aurez à utiliser dans l'interpréteur.

- Un dossier `course_language` qui contient la définition du langage (`ast.ml`), ainsi que tous les fichiers concernant ce langage (pour l'instant, uniquement le fichier `interpreter.ml`). Pour `ast.ml`, vous pouvez vous contenter d'en regarder le `.mli` (ou la doc générée), le code contient juste des fonctions d'affichage. Vous avez à compléter le fichier `interpreter.ml`.

Comme ce sera toujours le cas, vous avez à votre disposition un `Makefile` qui génère un exécutable `ast_interpreter.out` et la documentation, que vous pourrez consulter via le fichier `doc.html`.

Dans un premier temps, vous pouvez utiliser cet exécutable pour afficher les noms des exemples disponibles, ainsi qu'un affichage en forme de code lisible des exemples qui sont à votre disposition. Une fois votre interpréteur implémenté, vous pourrez grâce à cet exécutable, exécuter ces exemples. Tapez `./ast_interpreter.out -help` pour avoir le message d'usage. L'option `-list` vous donnera la liste des exemples disponibles. Certaines expressions ou instructions utilisent des variables qui ne sont pas affectée dans le bloc de code. Vous pouvez leur donner une valeur avec la ligne de commande en fournissant au programme un argument au format "$<$ *nom* $>:<$ *valeur* $>$". Par exemple, si on lance `./ast_interpreter.out exo2 x:4 y:1.2`, l'expression nommée exo2 sera évaluée dans un environnement où x vaut l'entier 4 et y le flottant 1.2. Il est possible de rentrer tous les types par ce biais, y compris des tableaux (avec la syntaxe [1,2,3] (par exemple)). Pour les programmes complets qui disposent d'arguments, on mettra la valeur au même format sans préciser leur nom. Par exemple `./ast_interpreter.out print_args true [1,2,3]`.

Vous pouvez également charger le projet dans `utop` (avec `dune utop` dans un terminal) si vous voulez tester vos fonctions indépendamment ou le faire sur d'autres exemples à la volée, ou encore si vous avez besoin de la forme "arbre" des programmes.

Vous avez à compléter le fichier `interpreter/interpreter.ml`.

Il contient l'interpréteur à compléter entièrement. Les plus grosses fonctions sont `interpret_expr` et `interpret_instruction` qui interprètent respectivement les expressions et les instructions. Elles sont mutuellement récursives à cause de l'expression d'appel de fonction, (les autres expressions n'utilisent pas les instructions). Il vous est fortement suggéré de faire un *pattern matching* sur l'expression/instruction à interpréter et de terminer par un cas par défaut (avec `_`) qui renvoie une erreur pour les cas que vous ne traitez pas encore.

Il vous est conseillé de développer votre interpréteur en respectant l'ordre des sous-langages suivants :

- LCalc
- LBranch
- LArray
- LFunc

Cet ordre vous donne une difficulté croissante (et les exemples des derniers langages utilisent les constructions des précédents...).

Ce TD (et même cet exercice) est prévu pour plusieurs semaines donc ne vous pressez pas, l'essentiel est de bien comprendre ce que vous faites.

Vous aurez à rendre votre fichier `interpreter.ml` d'ici quelques semaines (date à préciser) sur le moodle du cours.

Exercice 3: Pour aller plus loin Si vous avez terminé l'implémentation de l'interpréteur de notre langage, et qu'il se comporte en accord avec la sémantique, nous pouvons rajouter des constructeurs à notre langage, ou changez certains comportements.

Pour cela, il vous faudra modifier le module ast/ast.ml et ast/ast.mli (les deux): essentiellement, vous ajouterez des constructeurs (cas) aux types représentant les programmes, et implémenterez les cas correspondants.

Quand vous en êtes là, discutez-en avec votre chargé de TD, mais voici quelques idées, par ordre de complexité :

- Permettre le passage des cases de tableau par référence comme argument de fonction.
- Remplacer la stratégie de passage par référence de `var` par une stratégie de copy-restore (i.e., la valeur est copiée avant l'appel, puis la valeur de fin est répercutée ensuite – le comportement changera par rapport au passage par référence uniquement si une case mémoire est passé deux fois comme argument à une même fonction).
- Ajouter un type pointeur au langage et ajouter les expressions pour le lire et l'affecter. On n'ajoutera pas d'arithmétique des pointeurs (ce qui serait plus complexe).
- Ajouter une instruction permettant de déclarer une fonction locale.
- Ajouter la possibilité de passer des fonctions comme arguments d'autres fonctions.

Pour toutes ces extensions, pensez à créer des programmes pour tester ces extensions. Vous pourrez les ajouter au programme en modifiant le fichier `examples_generated.ml` du dossier `examples` (en les ajoutant aux trois listes de programmes).