Compilateur PtiCaml

troisième partie : analyse sémantique

à rendre avant le 8 décembre 10 heures

Le but de cette dernière partie consiste tout d'abord à produire du code exécutable *correct* à partir d'un fichier PtiCaml.

1 Sémantique de PtiCaml

La sémantique de PtiCaml est celle de Caml. En particulier, l'ordre d'évaluation des arguments d'une fonction, des éléments d'une liste ou d'un n-uplet n'est pas spécifié. On ne cherchera pas non plus à libérer la mémoire allouée pour les clôtures, les listes et les n-uplets.

2 Production de code

2.1 Langage cible

La machine abstraite à pile servant de langage cible est une légère extension de la machine décrite et utilisée dans le cours. Elle est détaillée, ainsi que les interprètes qui en sont fournis, dans le document annexe *Machine virtuelle pour le projet de compilation.*

2.2 Organisation des données

2.2.1 Représentation des valeurs

Types primitifs. Les valeurs de types primitifs sont représentées de la manière suivante :

- les entiers et les flottants, par ceux de la machine virtuelle ;
- les chaînes de caractères, par celles de la machine virtuelle ;
- les booléens, par des entiers avec la même convention que la machine virtuelle : 0 représente `false` et 1 représente `true` ;
- les valeurs de type `unit`, par le même entier (0 par exemple).
- la constante `[]` par l'entier 0

N-uplets et listes. Les n-uplets et les constructeurs de liste (autre que `[]`) seront représentés par un pointeur vers une donnée allouée sur le tas. Selon cette représentation, le filtrage sur les listes sera donc réalisé en comparant la valeur filtrée avec 0, qui représente nécessairement `[]` (toute valeur allouée sur le tas sera un pointeur différent de 0).

Valeurs fonctionnelles. Une valeur fonctionnelle est représentée par une *clôture*, c'est-à-dire un bloc alloué sur le tas contenant d'une part un pointeur vers le code de la fonction et d'autre part les valeurs des variables libres de cette fonction.

2.3 Schéma de compilation

Conversion en clotûres. Étape d'identification des clotûres et de désimbrication. À l'issue de cette phase, le programme ne contient plus de fonctions imbriquées (cf. cours et TD).

Conventions des appels. Dans le cas d'une application $e_1 e_2$, l'appelant empile successivement :

1. une place pour la valeur de retour ;
2. l'argument e_2 ;
3. l'adresse du code de la fermeture associée à e_1.

Primitives. Pour ne pas avoir de conventions d'appels différentes entre primitives et clotûres, vous devrez générer du code représentant les clotûres pour chaque primitive de PtiCaml.

3 Opérateurs polymorphes.

Bien que les règles de typage permettent l'utilisation des opérateurs de comparaison à des valeurs de types quelconques, il est assez difficile de générer le code pour ces opérateurs. Aussi, pour ne pas doublement pénaliser ceux qui n'auront pas implémenté complètement la sémantique des opérateurs $=$, $<$, $<=$, $>$ et $>=$, les tests automatiques qui seront réalisés lors de la correction porteront uniquement sur des comparaisons d'entiers et de flottants.

Pour implémenter correctement la sémantique des opérateurs polymorphes, il faut ajouter un champ supplémentaire à chaque bloc alloué en mémoire afin de stocker la *taille* de la valeur : cette taille sera donc n pour un n-uplet, 2 pour un constructeur de liste et 0, par convention, pour une clotûre. Cette information sera utilisée par une procédure générique de comparaison (que vous devez implémenter) afin de réaliser une comparaison *structurelle* des données.

4 Travail demandé

Adapter votre compilateur pour qu'il effectue la production de code. Votre compilateur devra reconnaître les deux options existantes `-parse-only` et `-type-only` (cette dernière indiquant désormais de s'arrêter juste après la phase d'analyse sémantique).

En l'absence d'erreur lexicale, syntaxique ou sémantique, la production de code ne doit pas échouer, et le code doit être produit dans le fichier `fichier.vm` si le fichier source s'appelle `fichier.ml`.

Votre code devra comprendre au moins un fichier `compile.ml` contenant les principales fonctions de production de code. L'introduction d'autres fichiers devra être justifiée dans le rapport.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `compilo`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Votre projet est à rendre à votre chargé de TP par e-mail[1] au plus tard le 8 décembre à 10h00.

[1] `Sylvain.Conchon@lri.fr`, `Louis.Mandel@lri.fr`, `Romain.Bardou@lri.fr`