Projet du cours « Compilation »
Jalon 6 : Compilation de Hobix vers Fopix

version numéro 1.0

1 Présentation de Hobix

La syntaxe de Hobix est définie comme suit :

			Programmes
p	::=	$\overline{d}$	*Liste de définitions*
			Définitions
d	::=	vd	*Valeur*
	|	**external** f	*Symboles externes*
			Définitions de valeurs
vd	::=	**val** $x\ = e$	*Valeur simple*
	|	**fun** $\overline{fd}$	*Fonctions mutuellements récursives*
			Définitions de fonctions récursives
fd	::=	$f(\overline{x}) = e$	*Définition de fonction*
			Expressions
e	::=	l	*Litéraux*
	|	x	*Variable*
	|	**val** vd **in** e **end**	*Définition locale*
	|	$e(\overline{e})$	*Application de fonction*
	|	**fun** $(\overline{x}) \Rightarrow e$	*Fonction anonyme*
	|	**if** e **then** e **else** e **fi**	*Expression conditionnelle*
	|	**while** e **do** e **end**	*Boucle*

Cette grammaire correspond à un sous-ensemble de la syntaxe du module `HopixAST`. Les constructions ignorées pour le moment seront prises en charge plus tard dans le semestre.

Remarquez les différences entre Hobix et Fopix :

— En Fopix, les fonctions sont des valeurs de seconde classe, tandis qu'en Hobix, elles sont de première classe.

— En Fopix, les fonctions sont définies globalement, tandis qu'en Hobix, elles peuvent être le résultat d'un calcul ou introduites sous la forme de fonction anonyme.

— En Fopix, il y a des pointeurs de fonction et une séparation claire entre l'espace de noms des fonctions et celui des autres valeurs. En Hobix, il n'y a pas de pointeurs de fonction et pas de séparation entre les valeurs fonctionnelles et les autres.

Ces différences doivent donc être prises en charge par la passe de compilation de Hobix vers Fopix dont il est question dans ce jalon. Pour vous aider à le réaliser, on vous guide en décomposant l'écriture de cette passe en plusieurs étapes. Pour le moment, comme pour les précédents jalons, nous allons considérer seulement des programmes Hobix mettant en jeu des litéraux entiers.

L'essentiel de la transformation consiste expliciter des **fermetures** en Fopix. Ces fermetures vont servir à représenter les fonctions de Hobix. On rappelle qu'une fermeture est une valeur qui contient à la fois un pointeur de code et les valeurs des variables libres de ce pointeur de code.

1.1 Étape 1 : Comprendre Hobix

Pour bien comprendre HOBIX, commencez par étudier son analyseur syntaxique (pour comprendre sa syntaxe concrète) et son interpréteur (pour comprendre sa sémantique) en écrivant les programmes suivants dans ce langage :

1. Écrire un programme qui définit la fonction factorielle de façon récursive et qui affiche la valeur de 5!.
2. Écrire un programme qui définit une fonction `apply` qui prend une fonction `f` en argument ainsi qu'un argument `x` et qui applique `f` à `x`.
3. Écrire un programme qui définit une fonction `addk` qui attend un entier `k` et qui renvoie la fonction qui à `x` associe `x + k`.
4. Écrire un programme qui définit une fonction `curry` qui prend en argument une fonction f à deux arguments et renvoie une fonction qui attend un argument x pour produire une fonction qui attend un argument y pour renvoyer $f(x, y)$.
5. Écrire un programme qui définit deux fonctions mutuellement récursives f et g telles que

$$\begin{cases} f(x) = 1 & \text{si } x = 0 \\ f(x) = g(x-1) & \text{sinon} \\ g(x) = 0 & \text{si } x = 0 \\ g(x) = f(x-1) & \text{sinon} \end{cases}$$

Vous pouvez tester l'interprétation de vos programmes à l'aide de la commande :

```
flap -s hobix -d true -r true -VV true your-program.hobix
```

1.2 Étape 2 : Calcul des variables libres

Pour être capable de compiler une fonction anonyme HOBIX **fun** $(\overline{x}) \Rightarrow e$ en le code FOPIX qui construit une fermeture, il faut préalablement calculer l'ensemble des variables libres de cette fonction.

1. Qu'est-ce qu'une variable libre ?
2. Quelles sont les variables libres de **fun** $(\overline{x}) \Rightarrow e$?
3. Complétez la fonction `HobixToFopix.free_variables`.

1.3 Étape 3 : Les fonctions anonymes simples

La compilation d'une expression de la forme « **fun** $(\overline{x}) \Rightarrow e$ » produit 2 morceaux de code FOPIX distincts :

1. À la place de l'expression « **fun** $(\overline{x}) \Rightarrow e$ », on va trouver en FOPIX une séquence d'instructions qui :
   (a) Alloue un bloc de dimension suffisante pour contenir les valeurs des variables libres et le pointeur de code.
   (b) Initialise les cellules de ce bloc en plaçant en position 0 le pointeur de code vers la fonction FOPIX qui implémente le corps de la fonction anonyme (voir point 2 plus bas) et dans la suite des cellules les valeurs des variables libres. Notez qu'il faut se souvenir de la position de chaque variable libre dans la fermeture. (C'est le rôle du dictionnaire passé en argument à la fonction de compilation).
   (c) Retourne le bloc.
2. À "*toplevel*", une nouvelle fonction FOPIX est insérée. Elle attend les mêmes arguments $\overline{x}$ que la fonction anonyme et un argument supplémentaire qui est la fermeture elle-même.

1. Vous avez certainement remarqué que la compilation de HOBIX vers **Fopix** n'est pas purement locale : non seulement une expression HOBIX doit être traduite en une expression FOPIX mais cette traduction doit aussi produire un ensemble de définitions de fonctions à insérer avant la définition en cours de compilation. À partir de ce constat, expliquez le type de la fonction `expression` de `HobixToFopix.translate`.

2. Complétez le cas `S.Define` sans traiter les fonctions récursives pour le moment.
3. Complétez le cas `S.Fun`.
4. Pour le moment, contentez-vous de complétez le cas `S.Apply (a, b)` en le traduisant par un simple `T.UnknownFunCall (a', b')` où `a'` (resp. `b'`) est le code compilé de `a` (resp. `b`).
5. Observez l'effet de votre compilation sur les programmes 2, 3 et 4. Pourquoi la compilation de l'application est-elle incorrecte ?

1.4 Étape 4 : Les applications de fonction

L'application d'une fonction à une valeur en **Hobix** doit se traduire par une *auto-application* de la fermeture : on doit extraire le pointeur de code de la fermeture et lui passer les arguments effectifs de l'appel ainsi que la fermeture elle-même.

1. Corrigez le cas `S.Apply` de la fonction de traduction des expressions.
2. Testez votre passe de compilation sur les programmes 1, 2, 3 et 4.

1.5 Étape 5 : Les fonctions mutuellement récursives

Dans un premier temps, vous allez compiler les fermetures mutuellement récursives en suivant la même stratégie que l'interpréteur de Hopix écrit au premier semestre : une fonction récursive contient un pointeur vers elle-même et vers toutes les fonctions avec lesquelles elle est mutuellement définie.

1. Rappelez la suite d'instructions nécessaires à la création de fermetures mutuellement récursives.
2. Complétez la fonction `define_recursive_functions` de la fonction `translate` de `HobixToFopix`.
3. Testez votre passe de compilation sur le programme 5.

2 Travail à effectuer

La cinquième partie du projet est la conception et la réalisation de la traduction des programmes Hobix en programmes Fopix.

Le projet est à rendre **avant le** :

20 décembre 2019 à 23h59

Pour finir, vous devez vous assurer des points suivants :

— Le projet contenu dans cette archive **doit compiler**.
— Vous devez **être les auteurs** de ce projet.
— Il doit être rendu **à temps**.

Si l'un de ces points n'est pas respecté, la note de 0 vous sera affectée.

3 Log

23-11-2020 Version initiale.