Projet du cours « Compilation »
Jalon 4 : Compilation de Retrolix vers x86-64 (GCC)

version numéro 1.0

1 Présentation de Retrolix

La syntaxe de Retrolix est définie comme suit.

			Programmes
p	::=	$\overline{d}$	*Liste de définitions*
			Définitions
d	::=	**globals** $\overline{x}$ {b}	*Variables globales*
	|	**def** $f(\overline{x})$ {b}	*Fonctions*
	|	**external** f	*Symboles externes*
			Blocs
b	::=	**locals** $\overline{x} : \overline{I}$	*Bloc d'instructions*
			Instructions étiquetées
I	::=	$\ell : i$	*Instruction étiquetée*
			Instructions
i	::=	$r\,(r_1, \cdots, r_N)$	*Appel de fonction*
	|	**ret**	*Retour à l'appelant*
	|	$l \leftarrow \text{op}\ r_1, \cdots, r_N$	*Opération*
	|	**jump** ℓ	*Saut inconditionnel*
	|	**jumpif** $c, r_1, \cdots, r_N, \ell_1, \ell_2$	*Saut conditionnel*
			Comparaisons
c	::=	$GT \mid LT \mid GTE \mid LTE \mid EQ$	*Comparaisons*
			Opérations
op	::=	*Copy*	*Déréférencement*
	|	$Add \mid Mul \mid Div \mid Sub$	*Opérations arithmétiques*
	|	$And \mid Or$	*Opérations logiques*
			Valeurs affectables
l	::=	reg	*Registre*
	|	x	*Variable*
			Valeurs d'affectation
r	::=	lit	*Valeur litérale*
	|	l	*Valeur en mémoire*

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

Remarquez les différences entre Retrolix et x86-64 :

— un programme Retrolix peut utiliser un nombre arbitraire de variables locales et les registres x86-64, tandis qu'un programme x86-64 a un nombre limité de registres et doit utiliser la pile pour obtenir une zone de stockage temporaire ;
— le mécanisme de définition et d'appel de fonctions est présent en Retrolix mais en x86-64 ;
— l'évaluation d'un programme Retrolix revient à évaluer les multiples blocs d'instructions d'initialisation des variables locales. En x86-64, il n'y a qu'un point d'entrée, la fonction `main` ;
— certains litéraux (les chaînes de caractère, les booléens, les entiers sur 64 bits, les caractères) de Retrolix n'existent pas en x86-64.

Ces différences doivent donc être prises en charge par la passe de compilation de Retrolix vers x86-64 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, nous allons ignorer partiellement le dernier point en ne considérant seulement des programmes Retrolix mettant en jeu des litéraux entiers.

1.1 Étape 1 : comprendre Retrolix

Pour bien comprendre Retrolix, commencez par étudier son analyseur syntaxique (pour comprendre sa syntaxe concrète) et son interprète (pour comprendre sa sémantique) en écrivant les programmes suivants en Retrolix.

1. Le programme vide.
2. Un programme qui contient une variable globale `x` valant 6, une variable globale `y` valant 7, une variable globale `z` valant `y * x`, et une variable globale `k` valant `z - x`.
3. Un programme qui calcule 5! et stocke le résultat dans une variable globale `res`.
4. Un programme qui appelle la fonction externe `observe_int`, qui attend un unique argument entier, en lui passant la valeur 42.
5. Un programme qui (i) appelle une fonction externe `add_eight_int` qui attend huit entiers et en retourne la somme S et (ii) utilise la fonction `observe_int` pour observer S.
6. Un programme qui (i) définit une fonction nommée `fact` qui attend un argument n et qui renvoie $n!$ (en calculant cette valeur itérativement) ; (ii) appelle cette fonction pour initialisation une variable globale x ; (iii) utilise la fonction `observe_int` pour observer x.
7. Un programme similaire au précédent mais qui calcule la factorielle récursivement.
8. Un programme qui (i) définit une fonction nommée `add_eight_int` qui attend huit entiers et en retourne la somme S puis (ii) affiche la somme des entiers de 1 à 8 à l'aide la fonction `observe_int`.

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

```
flap -s retrolix -d true -r true your-program.retrolix
```

1.2 Étape 2 : compiler le programme vide

Un programme vide en Retrolix ne se traduit pas en fichier vide en x86-64. En effet, pour que `gcc` puisse produire un fichier exécutable, il faut lui fournir un programme x86-64 avec un point d'entrée nommé `main`. Il est aussi recommandé d'appeler la fonction `exit` de la `libc` pour sortir convenablement du programme. La génération de la fonction `main` a été implémentée pour vous.

1. Utilisez la commande

   ```
   flap -t x86-64 empty.retrolix
   ```

   pour compiler un fichier vide `empty.retrolix` vers un fichier `empty.s`. Vérifiez que le contenu de ce fichier vous semble raisonnable.
2. Pour compiler le fichier `empty.retrolix` vers un fichier exécutable, on peut soit invoquer GCC sur le fichier `empty.s` produit à l'étape précédente via la commande

   ```
   gcc -no-pie -o empty.elf empty.s,
   ```

   soit utiliser directement flap :

   ```
   flap -t elf empty.retrolix.
   ```

   Les deux commandes produisent un binaire exécutable `empty.elf`. Lancez-le.
3. Lisez le code de la fonction `generate_main` du foncteur `Codegen` dans `retrolixToX86_64`.

1.3 Étape 3 : gestion des variables globales et opérateurs arithmétiques

On souhaite maintenant compiler les programmes (2) et (3). Pour cela, il faut prendre en charge la lecture et écriture dans les variables globales, ainsi que les opérations *Copy* et *Mul*. Pour simplifier la traduction, on suppose que le registre `%r15` est réservé au compilateur.

1. Implémentez les fonctions `mov`, `mul`, et `sub` du module `InstructionSelector`.
2. Implémentez la fonction `location_of` du module `FrameManager`, en supposant que la variable passée en argument est toujours une variable globale (l'argument de type `frame_descriptor` n'est donc pas utilisé pour l'instant).
3. Compilez et testez les programmes (2) et (3). Pour cela, vous exécuterez les programmes exécutables obtenus à l'aide de `gdb` et observerez la valeur des variables globales au cours de l'exécution.

1.4 Étape 4 : Traduire les appels de fonctions

On s'intéresse maintenant aux programmes (4) et (5). Il faut donc maintenant implémenter les appels de fonctions. Votre programme RETROLIX doit donc respecter les conventions d'appel x86-64 et System V : les six premiers arguments doivent être passés à travers les registres `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` et `%r9`, et le retour doit être attendu dans le registre `%rax`. Il faut aussi s'assurer que les registres x86-64 non préservés par les appels de fonctions ne sont pas utilisés dans votre code RETROLIX. Enfin, les arguments à partir du septième doivent être passés dans la pile. Ces arguments passés dans la pile sont ceux modélisés par les arguments effectifs des appels de fonction de RETROLIX.

1. Implémentez la fonction `call` du module `FrameManager`. Vous pouvez ignorer l'argument `kind`, ainsi que le descripteur de cadre de pile (ce dernier sera utilisé dans les questions suivantes).
2. Écrivez un fichier `runtime.c` que vous compilerez avec GCC. Ce fichier contient les fonctions `add_eight_int` et `observe_int` suivant les descriptions des programmes (4) et (5). En ce qui concerne `observe_int(n)`, on se contentera pour l'instant de terminer le programme en renvoyant `n` comme code de retour. On rappelle qu'on peut afficher le code de retour d'un programme UNIX via la commande ci-dessous.

   ```
   ./monprogramme; echo $?
   ```

3. Compilez et testez les programmes (4) et (5). Vous générerez le code assembleur x86-64 avec `flap`, et compilerez avec la commande suivante :

   ```
   gcc -no-pie -o fichier.elf fichier.s runtime.c.
   ```

   Dans le cas (probable) où vous obtiendriez une erreur, utilisez `gdb` pour comprendre sa cause.

1.5 Étape 5 : Traduire les définitions de fonctions

Pour finir, on se concentre sur la traduction des définitions de fonctions. On rappelle que lorsqu'une fonction est invoquée, elle alloue un bloc d'activation sur la pile pour y stocker ses variables locales. Lorsque l'exécution de cette fonction est terminée, la pile doit être désallouée. Après avoir traité ce dernier problème, vous devriez pouvoir compiler les programmes (6), (7) et (8).

1. Implémentez les fonctions `frame_descriptor`, `function_prologue`, et `function_epilogue` du module `FrameManager`. Complétez `location_of` pour gérer le cas des variables locales et des paramètres, en utilisant l'argument de type `frame_descriptor`.
2. Implémentez les fonctions restantes dans `InstructionSelector`.
3. Compilez et testez les programmes (6) à (8). Pour cela, vous exécuterez les programmes exécutables obtenus. Dans le cas (probable) où vous obtiendriez une erreur, déboguez avec `gdb`.
4. Implémentez l'alignement de la pile, suivant l'ABI System V x86-64. Nous rappelons que `%rsp` doit être aligné sur 16 octets juste avant chaque instruction `call`. Le plus simple est sans doute d'enrichir votre fonction `FrameManager.call`.

   Pour tester votre respect de l'ABI, modifiez la fonction `observe_int` pour utiliser `printf`, et retestez tous vos programmes. Vous pouvez utiliser `gdb` pour vérifier manuellement l'alignement de `%rsp` à chaque appel de fonction.

2 Travail à effectuer

La quatrième partie du projet est la conception et la réalisation de la traduction des programmes Retrolix en programmes x86-64.

Le projet est à rendre **avant le** :

05 décembre 2022 à 19h59

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

14-11-2022 Version initiale.