PROJET COMPILATION

MANAL BENAISSA
GUILLAUME CAILHE
NADIA BENMOUSSA
FARIS BOULAKHSOUMI
FLORIAN ARGAUD
FABIEN LEFEBVRE

TYPECHECKING

Typechecking

Se fait en 3 étapes :

- Génération des types de variables
- Génération des équations de types
- Résolution de ces équations et validation du typage.

Typechecking – Génération d'équations

- On ne se sert pas de l'AST à ce moment, mais une fonction récursive qui s'applique sur chaque sous-expression du programme mincaml.
- Les équations sont stockées dans une arraylist (qui sera utilisée dans la 3ème étape)
- Les équations sont implémentés comme couple de type
- Les environnements sont implémentés comme une liste de variables associées à leur type.

Typechecking – Résolution des équations

- Unifier et résoudre les équations générés par l'algorithme *GenererEquations*(EnvironnementType env, Exp e, Type t)
- Un booléen "bienTypé" est mis à *false* ce qui signifie que le programme est mal typé

FRONTEND

K-normalisation

- Lors de l'étape de la K-Normalisation, notre but est de décomposer notre programme, initialement étant un ensemble d'opération, en une série d'opération. Cela se représente de la manière suivante :

  a+bc-*v* *devient par exemple :* *(a+(b*(c-v)))*

- Notre programme étant initialement en MinCaml, on applique le processus au MinCaml :

  Let x1 = 1 + 2 in x1 devient : Let x1 in Let v1 = 1 in Let v2 = 2 in v1 + v2 in x1

A-conversion

- L'alpha-conversion est le renommage des variables liées. Comme pour la K-Normalisation, nous avons utilisé le visiteur de l'AST pour effectuer cette étape. Il nous a fallu utiliser une HashMap pour gérer l'environnement courant et le modifier lors des let, var et letRec afin de gérer ce renommage.

Let-reduction

- Linéariser toutes les définitions de *let* (mais pas la let-rec)
- **Entrées :** Ast K-Norm et alpha converti
- **Sorties :** Ast K-Norm, alpha converti et let-réduit

ASML

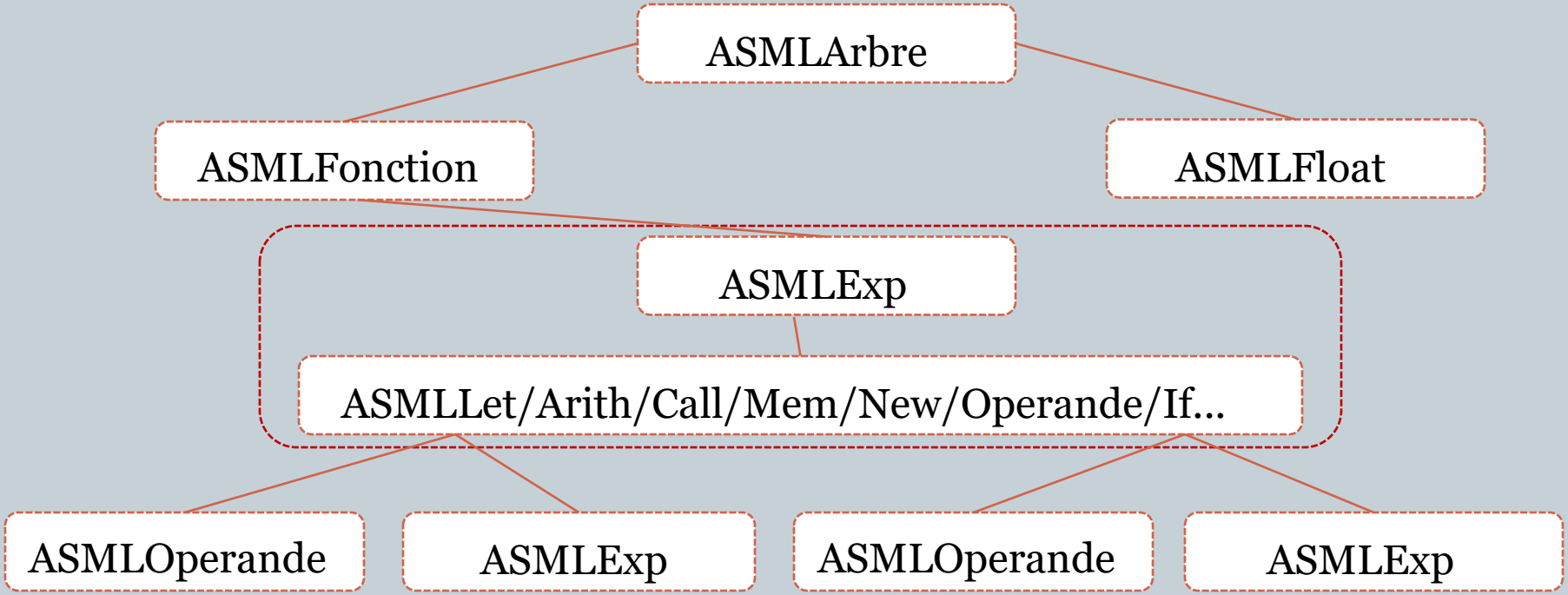
Dans cette partie on souhaite modifier une dernière fois notre AST pour fournir en sortie un fichier .asml :

- Implémentation de l'interface *ObjectVisitor* afin de modifier l'AST
- Gestion de tout les cas possibles dans les méthodes **visit** à redéfinir.
- Stockage des déclarations dans des attributs de type *string* pour pouvoir écrire dans le fichier .asml
- Écriture dans le fichier dans le bon ordre avec les déclarations de flottants, le reste des déclarations, puis le corps du programme.

BACKEND

Datatype

Nous utilisons un arbre syntaxique afin de représenter le code ASML, effectuer l'allocation de registre et générer le code ARM.



Allocation des registres

allocation basique (spill) :

- Les registres utilisés par les paramètres sont les registres de r0 à r3.
- Si il y a plus de 4 paramètres, les paramètres sont mit sur la pile.
- La dernière instruction de la fonction enregistre le résultat dans r0.
- Si il y a strictement moins de 8 variables locales, on utilise tout les registres de r5 à r10 pour les enregistrer.
- Sinon, on utilise les registres de r5 à r8 pour les 5 premières variables, et les autres variables vont sur la pile, et on se sert de r9 et r10 pour les charger quand nécessaire.
- r12 contient le résultat d'une expression.
- r11 est le frame pointer.
- r4 sert de pointeur de tas

Génération de code

La génération du prologue et de l'épilogue des fonctions est inspiré de celle généré par la commande : **arm-none-eabi-gcc -S ... -Oo**

Chaque instruction génère son propre code. Exemple (ASMLIf) :

- charger les variables de la comparaison
- tester la condition "<op de comparaison> <val1> <val2>"
- brancher si faux "<bne/bgt...> TAG_ELSE"
- code si vrai (généré par les ASMLExp filles et non le ASMLIf)
- brancher sur la suite du code "b TAG_SUITE"
- code faux :
  TAG_ELSE:
  <code else>
- code suite :
  TAG_SUITE: