Projet Compilation
M1 INF / MOSIG *BugPerture Science*
Compilateur Min-caml

Documentation Technique

Introduction :

Nous avons implémenté un compilateur du langage MinMl, une sous-partie du langage Ocaml. Ce compilateur est partiel – il transforme un programme MinMl en un programme en langage d'assemblage ARM.

Ce document présente les détails techniques de notre implémentation en Java.

Lors de notre processus de compilation, nous effectuons successivement les opérations suivantes :

- Lexing/Parsing
- Typecheck
- K-normalisation
- Alpha-Conversion
- Beta-Réduction
- Let-Réduction
- Inline Expansion
- Propagation des constantes
- Suppression des définitions inutilisées
- Allocation de variables
- Génération ARM
- 'Immediate' Optimisation

Structures utilisées :

Afin de préserver la cohérence des transformations successives, nous avons créé des structures intermédiaires d'AST pour les formes K-normale (afin d'interdire les valeurs immédiates en-dehors des let) et ASML (afin d'ajouter les closures et les accès mémoire).

Lexing/Parsing :

L'analyse lexicale et le parsing du programme MinMl nous était fourni ; nous sommes donc partis d'un arbre syntaxique abstrait fourni par le parser.

Typecheck :

Entrée : un AST
Sortie : un booléen et une table de correspondance symboles/types

Nous avons suivi l'algorithme fourni, en deux étapes :
- Nous génrérons d'abord les équations de types en parcourant l'AST
- Puis nous résolvons le système d'équation génré. Si le système aboutit à une incohérence, le programme est déterminé mal typé et une erreur est renvoyée. La résolution d'un tel système est facilitée par le caractère monomorphique du langage.

Au cours de ces deux étapes, nous remplissons également une table de correspondance entre symboles (identifiants) et types, qui servira pour les futures opérations.

Remarques/choix d'implémentation :
Nous considérons possibles les comparaisons entre tous les types. Néanmoins, nous n'implémenterons de telles opérations que sur les entiers.

K-Normalisation :

Entrée : un AST
Sortie : un AST K-normal

Nous avons suivi l'algorithme fourni afin de transformer un AST en un AST K-Normal.
Le principe de l'arbre K-Normal est de n'avoir aucune valeur immédiate dans les opération, c'est à dire qu'à part les nœuds Let , LetRec, LetTuple et If, tous les autres nœuds ont uniquement des Vars en fils. Ainsi cette étape créé de nombreuses variables supplémentaires , mais permet d'avoir un arbre avec des operations 3 adresses, ce simplifie la tache de la traduction en ARM.

Alpha-Conversion :

Entrée : un AST K-normal (et éventuellement un mapping initial de renommage)
Sortie : un AST K-normal et Alpha-converti

Nous avons suivi l'algorithme fourni, afin de renommer les variables en double dans le programme. En réalité, nous renommons toutes les variables non externes à l'AST (fonctionnalité utilisée dans l'Inline Expansion). Pour une commodité de lecture dans les opérations suivantes, nous créons des identifiants '?vx' pour les variables et '?fx' pour les fonctions.
Pour convertir un programme entier, le mapping initial est vide – à l'exception des fonctions de bibliothèque (comme print_int). Ces fonctions de bibliothèques sont accessibles par la classe MinMlLibrary.
La possibilité de fournir un mapping initial est utilisée dans l'Inline Expansion.

Bêta-Réduction :

Entrée : un AST K-Normal et Alpha-converti
Sortie : un AST K-Normal, Alpha-converti
La Bêta-Reduction est une opération qui permet simplement la suppression de variables intermédiaires inutiles. Ainsi les expressions de la forme :
Let a = 2 in let b = a in b + 2
sont remplacées par Let a = 2 in a + 2
Cette étape doit être effectuée plusieurs fois pour permettre la réduction complète, aussi tant que la bêta-reduction effectue une modification dans l'arbre, celle-ci est recommencée jusqu'à stabilisation.

Let-Réduction :

Entrée : un AST K-normal et Alpha-converti
Sortie : un AST K-normal, Alpha-converti Let-réduit

Le processus de let-réduction « aplatit » les let imbriqués de la forme :
let x =
    let y = e1 in e2
in e3
en

let y = e1 in
let x = e2 in
e3
L'opération est simple et préserve les propriétés de K-normalisation et d'Alpha-conversion.

Inline Expansion :

Entrée : un AST K-normal et Alpha-converti, et une taille maximale de fonction
Sortie : un AST K-normal et Alpha-converti

L'Inline Expansion consiste à copier les corps des petites fonctions à la place de leurs appels. Cette opération seule ne préserve pas la propriété d'Alpha-conversion. Nous effectuons donc sur chaque copie du corps de la fonction une Alpha-Conversion, avec en mapping initial un mapping des arguments formels vers les arguments effectifs.
Les fonctions récursives ne sont pas copiées – le programme deviendrait trop volumineux.
De plus, cette opération ne préserve pas la propriété de Let-réduction ; nous effectuons donc une nouvelle Let-réduction après cette opération.

Propagation des constantes :

Entrée : un AST K-normal et Alpha-converti
Sortie : un AST K-normal et Alpha-converti

La propagation des constantes est une optimisation permettant de calculer statiquement une partie des valeurs immédiates du programme.
Cette opération est simple si on ne conserve pas la propriété de K-normalisation – cela crée beaucoup de valeurs immédiates en-dehors des let, ce qui est problématique.
Nous maintenons donc une valeur booléenne lors du parcours de l'AST, qui indique si la propagation doit être « totale » à tel ou tel endroit ; si oui, les valeurs immédiates sont remontées le plus haut possible ; si non, un nouveau let est créé pour préserver la propriété.

Suppression des définitions inutilisées :

Entrée : un AST K-normal et Alpha-converti
Sortie : un AST K-normal et Alpha-converti

Les deux optimisations précédentes rendent inutiles beaucoup de définitions. Cette opération est dédiée à la suppression de telles définitions.

Nous avons choisi d'autoriser plusieurs passages sur les optimisations, plutôt que de trop développer chacune : en effet, l'Inline Expansion augmente la taille du programmes, et la Suppression des définitions le réduit ; alterner ces optimisations est donc plus efficace.

De plus, afin de générer une nouvelle table de correspondance entre symboles et types après ces opérations, nous effectuons un nouveau Typecheck.

Génération des Closures et de l'ASML

Entrée : un AST K-normal et Alpha-converti
Sortie : Un AST ASML

La transformation de l'AST K-normal et Alpha-converti en AST ASML est fait de manière

automatique .On extrait chaque fonction et float du programme et on l'insert dans la liste des fonctions puis on met en fin de liste le main .En même temps la génération des closures est fait en fonction des besoin (si la fonction retourne une fonction on met le retour dans une closure, si la fonction accède à des variables ou des fonctions qui ne sont pas définit dans son environnement on créé une closure qui contient la fonction et tous les paramètres utilisés qui ne sont pas définies dans l'environnement)

Allocation de variables :

Entrée : Un AST ASML
Sortie : Un AST ASML

a) Affectation de variables

Nous somme partis sur une stratégie simple d'allocation de variables : toutes les variables rencontrées sont allouées dans les registres, puis dans la pile si il n'y a plus de registre disponible. L'affectation est alors enregistrée dans la table de symbole associée à l'environnement de la variable (fonction où elle est déclarée)

Convention utilisée pour l'allocation dans les registres :

R0-R3 , R6-R10, R12	Registres diponibles pour les allocations de variables
R4, R5	Registres réservés pour les opérations de calculs
R11	fp
R13	sp
R14	lr
R15	cpsr

b) Affectation de labels aux fonctions

Pour chacune des fonctions du programme, on crée sa table de symbole récursivement à l'aide de l'alllocateur, et on lui assigne un label correspondant à son nom. La table de symbole créée sera accessible dans la structure Label_fonction ainsi que son identifiant.
Le label est ensuite stocké lui-même dans la table de symbole de l'appelant.

c) Création de tables de symboles

La table de symbole est le principal élément traité dans cette partie. Toutes les variables et tous les labels y sont répertoriées ainsi que leur «position» dans la mémoire.

Ceci introduit la notion de «position» décrite par la classe Position_variable. Une position dans la table de symbole représente concrètement l'endroit de la mémoire où doit être stocké ou récupéré l'objet. On distingue notamment :

- Position_registre : indique que la variable est stockée dans un registre contient le numéro du registre en question
- Position_pile : indique que la variable est stockée dans la pile, contient l'offset par rapport au fp de l'environnement
- Label_Fonction : indique le nom d'une fonction, et contient aussi la table de symbole qui lui est associée
- Label_float : indique un floatant qui sera à allouer dans le tas.
- Position_retour : désigne pour une fonction, la position où sera stockée son retour

L’arbre ASML en entrée de l’allocateur est modifié pour permettre au générateur ARM de traiter plus facilement certains cas de figure :
- L’identification du retour d’une fonction, ainsi que sa récupération
- L’identification des arguments passés en paramètres à une fonction lors d’un call, dans sa table de symbole
- Allocation de valeur immédiates particulières (big integers) dans une position label

Génération ARM :

Entrée : Un AST ASML
Sortie : Un programme en langage d'assemblage ARM

Cette opération est réalisée par deux modules qui communiquent :
-Un visiteur, qui va parcourir l’arbre ASML, et qui va appeler le second module :
-Un 'printer' ARM.

a) Le visiteur

Le visiteur sert de guide, il parcourt l’arbre et le modifie pour le faire écrire dans le bon ordre :
• Il ignore les expressions sans effets de bord.
• Il modifie l’arbre (notamment dans les if then else) pour relier des expressions qui seraient trop éloignées (notamment les expressions de la forme Let x = if)

A la sortie de ce module, l’arbre sera réduit à une expression sans effets de bord.

b) Le printer ARM

Le printer écrit le code ARM attendu. Il se sert uniquement de la table de symboles pour charger les données, prévoir les registres de destination et faire les load & store nécéssaires.

Il gère les expressions usuelles : Let – Call – CallClosure – Put

De plus, c’est également par son biais qu’on va mettre en place l’environnement, tant des fonctions que du main.

Conventions prises :
Pour plus de simplicité, deux registres sont reservés pour faire toutes les opérations, notamment celles entres des variables étant stockées en pile ou en label.

Dans le cadre des closures, le pointeur vers l’espace mémoire de la closure est toujours passé en premier argument (= R0).

'Immediate' Optimisation :

Ce processus n'est plus une optimisation à ce stade, il sert principalement à vérifier qu’une valeur immédiate est possible à écrire dans une instruction.

Il se base sur une traduction en binaire, suivi d’un parcours pour remarquer les longues suites de 0 ou de 1 nécéssaires a la possibilité de génération directe.

Rappel : Les valeurs immédiates sont codées sur 12 bits en ARM:

8 bits pour la valeur, et 4 bits pour un décalage circulaire (multiplié par deux, pour une rotation maximale de 30)
Les valeurs possibles sont ainsi de ce type :

une plage de 8 bits « significatifs », et une plage de 24 bits uniformes.

-décalé de 16 :
0x00 FF 00 00

-décalé de 24 :
0x F0 00 00 0F (plage séparée en deux par rotation jusqu'à la limite)

Il permet donc aux nombre qui ont une séquence d'au moins 24 '0' ou 24 '1' (via le MVN (caché par le compilateur ARM)) d'être écrits.