Compilation

MIni MOdules

Yoann Bourse

2009-2010 : Semestre 1

Résumé

Ce papier décrit la démarche suivie dans le cadre du projet MIMO réalisé pour le cours de compilation. J'insisterai sur les difficultés rencontrées et les solutions mises en place en suivant le plan de fonctionnement du compilateur. Faute de temps, le projet n'a pas pu être mené totalement à terme. Je décrirai donc l'étendue des fonctions de la version remise, ainsi que les problèmes connus à ce jour. Tout devrait marcher à l'exception de la production de code liée aux modules complexes, à cause de la défonctorisation.

Table des matières

1 Analyse lexicale et syntaxique

1.1 Syntaxe abstraite

La syntaxe abstraite est définie dans **syntax_tree.ml**. Elle suit les règles imposées de manière assez fidèle, dans un soucis de lisibilité. Une "surclasse" **rexpr** (pour Real Expression) permet de repérer la position de chaque expression afin de le préciser dans les rapports d'erreurs. La position est également enregistrée dans les déclarations et les modules. Les types sont encore sous leur forme de *string* : le typage devra les convertir. On distingue également variables et appels de fonctions, qui sont spécifiés sous la forme d'expression régulière dans l'énoncé du devoir.

Contenu non implémenté :
C'est la seule occurence de *Open*, qui est correctement parsé mais dont je n'ai absolument aucune idée de comment effectuer le typage et a fortiori la production de code

1.2 Analyse lexicale

L'analyse lexicale est compatible avec les commentaires imbriqués, et prend en compte les retours à la ligne pour repérer les positions des erreurs. Les symboles - unaire et binaire ne sont pas encore différenciés. Il a fallu faire attention à ne pas définir de mots clés supplémentaires (*ref* par exemple), ce qui aura une incidence sur les phases ultérieures.

1.3 Analyse syntaxique

A cette phase, de nombreux conflits se sont posés. Il a été crucial de définir les opérateurs binaires avec la fonction *inline*, ce qui en résolvait la plupart. Les autres problèmes étaient dues à une mauvaise hierarchisation primaire, qui a été corrigée par la suite. Les associativités ont en revanche posé peu de problème (sauf pour le point).
Cette opération extrapole les sucres syntaxiques (tels que *if ... then ...* extrapolé par *else ()*). Les options complexes de Menhir pour définir des listes avec séparateur sont peu utilisés par soucis de lisibilité. Au coeur de cette analyse se trouve la syntaxe que nous utiliserons pour les variables et les fonctions dans les modules, appellée *path* :
(**arborescence :** *string list*, **nom :** *string*)

2 Typage

Le typage a été une phase extrèmement complexe que je me suis efforcé de traiter utilement : la signature de chaque déclaration et de chaque module

est renvoyée dans le terminal à l'execution, à la manière de Ocaml. Cette voie simple et légère a été préférée à la fastidieuse écritude dans un fichier annexe. Cette phase utilise également de nombreuses exceptions afin de soulever des erreurs aussi précises que possibles pour aider l'utilisateur à corriger son programme, ce qui était ma priorité.

2.1 Type générique

Les types sont traduits de *string* vers le type caml *rtype* (real type). Le type générique est représenté par le rtype **Empty** nommé ainsi car il est principalement utilisé pour la liste vide. La généralisation est permise par une fonction *equal* qui vérifie si deux types sont égaux en prenant ce type vide en compte. De plus, la fonction de typage d'expression doit renvoyer un environnement afin de pouvoir réagir à la spécification d'un type pour une variable qui était auparavant de type vide.

2.2 Schema

Les fonctions sont représentés par leur schema, c'est à dire leur signature, sous une forme qui facilite le repérage du type de sortie : ainsi une fonction de signature $a \leftarrow b \leftarrow c \leftarrow d$ aura pour schéma : *(d,[a ;b ;c])*.
Les variables globales sont également enregistrées sont la forme de fonctions de schéma *(a,[])*.

2.3 Environnement

Toutes les variables et fonctions définies sont enregistrées dans l'environnement. La hierarchisation en modules a necessité un environnement plus complexe qu'une simple *string map*. Ainsi, un environnement a deux composantes : un *string map de schémas*, dictionnaire des fonctions et variables qui se trouvent à la racine courrante, et une deuxième composante, qui est soit *NoMod* s'il n'y a pas de module défini, ou un *string map d'environnements* qui permet d'accéder aux environnements des modules et de leurs sous modules.
Par exemple, *X.f* sera à la racine (1ère composante) de l'environnement repéré par X dans le dictionnaire de modules (2ème composante) de l'environnement initial.

2.4 Modules

Pour le typage des modules, il était necessaire de garder à la fois les variables et fonctions définies dans le module mais également la signature des arguments qu'il attendait. Ces arguments sont codés par le type *verifsig* (verification signature) qui contient un *string* (le nom formel de l'argument) et une *string map de schémas* qui représente les fonctions attendues. Un

module se compose donc d'une liste de *verifsig* pour ses arguments et d'un environnement qui est l'environnement résultant de l'évaluation de son contenu. Ainsi, lorsqu'un module est implémanté, il suffit d'ajouter cet environnement à l'environnement en cours. L'ensemble des modules est conservé dans un dictionnaire de modules dont les entrées sont leurs noms.

3 Traitement

Cette phase consiste à attribuer un nom unique à toutes les fonctions et à procéder à la défonctorisation, afin de ne plus avoir que des déclarations et d'avoir des étiquettes uniques dans l'assembleur produit.

3.1 Fonctions

Le principe est d'attribuer à chaque fonction un unique identifiant numérique qui sera incrémenté à chaque redéfinition, en laissant bien sur une exception pour les fonctions de base (ref, print_int...) qui sont compilées différement et n'ont donc pas d'identifiant avant leur première redéfinition par l'utilisateur. Pour se faire, chaque fonction voit son contenu et son identifiant conservé dans un *string map* qui sera transporté tout au long du programme. Une référence globale contient la version la plus récente de ce dictionnaire afin qu'il puisse survivre aux retours dans la récursivité.
Les *"path"* sont dépliés, si bien que les noms prennent la forme "module1ssmod3fun4". Toutes les fonctions utilisées ont d'ailleurs un argument *pref* qui permet d'ajouter un préfixe avant les noms des fonctions et variables.

3.2 Défonctorisation

De la même façon que pour les fonctions, les modules se voient attribués un numéro qui est stocké avec leur contenu dans un dictionnaire. A chaque implémentation du module, on recopie le contenu et on incrémente ce numéro. Les arguments du module prennent le nom des arguments formels (voir problème ci-dessous) afin que le code du module à proprement parlé reste le même (à l'exception des identifiants correspondant auxdits arguments formels). Pour ce faire, un dictionnaire supplémentaire en référence global associe à chaque module la liste de ses arguments formels pour pouvoir y accéder facilement.

3.3 Problèmes connus

Cette phase est celle qui a le plus souffert de l'ampleur du projet pour une seule personne car elle a été réalisée en dernier. Si elle fonctionne sans problèmes sur certains exemples relativement complexes (voir l'exemple

fourni), les bugs rencontrés sur de plus gros exemples laissent penser qu'elle mérite probablement d'être retravaillée en profondeur à partir de 0. Les dictionnaires, plutôt que d'être passés en arguments, pourraient par exemple devenir définitivement des références globales. L'inconvénient est que l'imperfection de cette phase handicape lourdement la compilation de modules.

Contenu non implémenté :
Bug connu : le programme ne renomme pas correctement les arguments de deux modules du même nom, comme par exemple :
M = Pascal(H)
N = Pascal(V)
En effet, le compteur de l'argument formel fictif *(X)* ne s'incrémente pas correctement

4 Production de code

A l'exception des modules (voir ci-dessus), cette phase devraitse dérouler sans erreurs.

4.1 Architecture MIPS

Les informations sur le processeur MIPS sont dans le fichier **mips.ml**. La principale difficulté à ce niveau a été le support des opérations *div* et *mod* qui ont nécessité un traitement particulier via une nouvelle instruction *Arith64* à cause de leur traitement particulier par le processeur MIPS. En effet, ce dernier, lors d'une division, enregistre le reste et le quotient dans un registre particulier 64 bits, où il faut ensuite aller chercher le résultat dont nous avons besoin.

4.2 Production de code

- Même si quelques opérations utilisent les registres, ce compilateur utilise à outrance la **pile** pour y enregistrer les résultats intermédiaires. En particulier, chaque compilation d'expression se termine en insérant la valeur en haut de pile. De nombreux tests ont été effectués avec le simulateur MIPS en java "Mars" qui permet une visualisation claire de la pile.

- Les **variables globales** sont stockées sur le tas, une table de hachage contient leur position par rapport à GP. C'est là aussi que sont enregistrées listes (deux cases mémoire : valeur et pointeur suivant) et

références (une case mémoire pour la valeur).

- Les **variables locales** sont conservées dans un environnement, simple dictionnaire de positions, et sont repérées par rapport à FP.

- Les **fonctions primitives** sont définies juste après le label main, et le code vient ensuite. Ceci bien entendu à l'exception de la fonction ref qui est un cas à part puisque son exécution modifie la taille du tas (et non la position de GP), conservée dans une référence globale.

- Les **pattern matching et les tests conditionnels** sont effectués grace à des étiquettes partageant un compteur commun pour l'unicité des étiquettes. On a dans chaque cas un saut (soit on saute au cas "faux" ou "liste vide" par un *jeqz*, soit on saute par dessus ce cas directement à la fin). L'utilisation d'underscore _ permet de s'assurer que l'on n'utilise pas des label déjà utilisés par des fonctions.

Contenu non implémenté :
Les opérations booléennes ne sont pas encore parresseuse. Cette implémentation n'a pas pu être effectuée faute de temps.

4.3 Compilation des fonctions

Le point délicat de cette phase est la compilation des fonctions. Avant chaque appel, RA et FP sont empilés pour être sauvegardés, puis les arguments de la fonction sont empilés dans un ordre précis. La fonction appellée (correspondant à un label) dépile ses arguments et s'execute, empilant son résultat final avant de retourner à l'adresse de retour. Il faut alors restaurer les anciens RA et FP, en n'oubliant pas de conserver le résultat et de surveiller la position de SP.

5 Execution

Le makefile semble correct, y compris pour la fonction clean. Le fichier produit est "mimo". Les modes -parse-only et -type-only sont implémentés et ne devraient poser aucun problème.