Compte rendu final du projet de compilation

Élie Gédéon et Guillaume Lagarde

Table des matières

1 Le Langage

1.1 Les instructions

Il s'agit d'un Pascal étendu. Il y a deux types de variables simples, les entiers et les booléens (tous deux stockés sur 4 octets), et deux types de variables composées, les tableaux et les références (pointeurs).
Le tableau est ici un type primaire ; on peut donc affecter des tableaux, renvoyer des tableaux ... On peut faire des tableaux de tableaux, des tableaux de booléens, des tableaux d'entiers, mais pas de tableaux de références. Pour construire un tableau, il faut faire Array of [a1..a2,b1..b2,etc..] of genre (genre étant quelconque, y comprit un tableau).

On ne peut faire que des références d'entiers ou de booléens.
Des primitives permettent la manipulation de références :

- !(exp) pour la valeur contenu à l'adresse expr.
- &(tableau) pour mettre l'adresse du tableau dans la référence.
- @(exp) pour convertir exp en référence.
- ^(exp) pour convertir la référence en entier. (utile pour enregistrer une référence dans un tableau)

On peut également ajouter des entiers aux références ; ajouter n à une référence revient à ajouter n*4 octets.

Pour l'allocation dynamique, nous avons les deux fonctions "pascal" suivante :

- new(expr) qui alloue une bloc mémoire de taille expr (la fonction nous renvoie le pointeur du début du bloc)
- free(pointeur) rend libre le bloc occupé à l'adresse donnée en argument.

Le compilateur supporte les structures de base, boucles for, while, repeat until, le if..then..else.. ;, le if..do.. ; (évite l'ambiguïté avec if..then..else.. ;), les affectations
Pour les entrées sorties, write liste_expression affiche les éléments de liste_expression les uns à la suite des autres ; on peut donc marquer write true,tableau,reference ; writeln ajoute un retour chariot final. read attend un entier sur stdin, et le renvoie.

On peut faire une déclaration anticipée d'une fonction en substituant son corps et ses déclarations internes par le mot-clé forward. Notez également le fait qu'à la déclaration d'une procédure ou d'une fonction sans paramètre, il ne faut pas mettre de (). (de la même manière qu'on note PROGRAM main ;)

Les commentaires se notent {commentaires} ou "commentaire". On peut imbriquer l'un dans l'autre ou l'autre dans l'un, mais pas plus.

Nous invitons l'utilisateur à lire les fichiers d'exemples.

1.2 Généralitées sur les fonctions

Une fonction peut appeler n'importe laquelles de ses filles, soeurs, tantes, grands-tantes, etc.
Les fonctions mutuellement récursives sont implémentées grâce à la déclaration anticipée.
De la même manière qu'en pascal, on peut passer des paramètres en référence. Seules les variables locales sont acceptées dans ce cas.

2 Choix et structures

L'idée phare de notre projet est l'utilisation d'AST, marquant chacun une étape importante dans l'avancement du projet. De nombreux aspects positifs se sont présentés avec cette attitude : tout d'abord, il y a eu un cloisonnement important entre les différentes parties de notre compilateurs, ce qui nous a permi de travailler à deux de manières efficaces et partagés. L'un pouvant travailler sur la partie pré-ast, l'autre sur la partie post-ast. Ceci nous a également autorisé à faire abstraction de certaines choses, comme l'idée de pile qui n'apparait pas à l'analyse de vivacité, et en cela, ce choix de structures en ast était bonne. Cependant, cela nous a aussi obligé à faire beaucoup d'administratif lors de la programmation, car les nombreuses transformations d'arbres en arbres oblige de grand matching coûteux en temps passé à programmer. Une structure plus élégante et moins redondante aurait été heureuse pour un compilateur futur.
Voici ci-après un descriptif de chacun de ces AST, ainsi que la description en détail de l'analyse de vivacité.

2.1 Ast

2.1.1 Description succinte

Cette structure représente juste ce que le analyseur grammatical renvoie.

2.2 Ast2

2.2.1 Description succinte

Cette structure est celle issue de l'analyse statique. Elle décrit de manière inductive le code.

2.2.2 Structure

L'idée générale est les définitions inductives ; aucune trace ici de table de hachage, la structure reste « propre ».

Une fonction est constituée d'un corps et d'une liste de fonctions filles. Ceci permet de faire certain traitement sur la fonction plus facilement, dans la mesure ou les fonctions filles sont directement accessibles.
Le code est lui aussi inductif. Une conditionnelle est donc constituée d'un booléen à évaluer ainsi que de deux listes d'instructions, suivant la direction prise. Petite particularité ; le TantQue a trois membres : le booléen à évaluer pour décider s'il faut continuer à itérer, le corps, et un bloc à exécuter avant l'évaluation du booléen. Ceci permet de déplacer le moment du test, permettant de faire des calculs avant.
Les constantes entières ou booléennes ont été mises dans le code. Un tableau de tableau représente les tableaux de constantes, et un tableau de string les tableaux de chaines. Une variable est identifiée par un couple, le premier membre représentant de combien on doit remonter (0 fonction courante, 1 mère, 2 grand mère...) ; le second est simplement le numéro de cette variable dans la fonction ascendante.

2.2.3 Choix

Un choix important est le suivant : il n'y a plus de fonction, un appel de fonction n'est qu'un appel de procédure auquel on passe la valeur de retour par référence. Il faut donc éventuellement faire un appel de procédure avant le calcul d'une conditionnelle ou d'une boucle, raison d'être de ce TantQue bizarre.
Aucune trace non plus de passage par valeur ou référence. L'idée est que toutes les variables sont passées par référence, quitte à faire une copie locale. L'idée est d'homogénéiser, le surcoût éventuel étant résorbée par l'analyse de vivacité, à même de faire disparaître ces transactions inutiles.
Pour identifier une fonction, le choix initial était de ne pas avoir d'identifiant particulier, en misant sur l'aspect inductif. Nous sommes revenus sur ce choix, principalement parce qu'il était difficile de construire des tables de hachages avec des fonctions en tant que clé. Une fonction est maintenant identifiée de manière unique par un chemin, c'est-à-dire une liste d'entiers.

2.3 Ast3

2.3.1 Description succinte

L'ast3 est la structure de donnée que l'on récupère suite à la génération de code intermédiaire. Elle se distingue de l'ast2 par la linéarisation des fonctions principales.

2.3.2 Structure

Les données manipulées sont des registres virtuels en nombre illimité de type int qui se distinguent des sauvegardes qui eux sont les registres prove-

nant d'une fonction mère ou les registres d'entrée sortie (IO). Les instructions qui subsistent sont désormais élémentaires comme les Jump, CJump,Appel de procédure, affectations de registre à registre, de sauvegarde à registre, de registre à sauvegarde, de tableau à registre, de registre à tableau ou encore des affectations binaire, c'est à dire décorés d'un opérateur binaire de type "r1 <- r2 binop r3" avec r1,r2,r3 des registres.

2.3.3 Choix

Nous avons choisis de différencier registres et sauvegardes dans l'idée de faciliter l'analyse de vivacité et l'allocation des registres, en effet on ne prévoit aucune analyse de vivacité sur les registres mères ou les registres IO.

2.4 Ast4

2.4.1 Description succinte

Cette structure représente le code avec allocation de registres. Il est donc très proche du code final.

2.4.2 Structure

Pour faire abstraction de la pile, un type particulier de registres a été défini : les registres de sauvegarde. Quand l'allocation de registres manque de registres, elle les place dans un registre virtuel. Charge à la génération de code de transcrire ça en mouvement de pile.
Quand une variable est utilisée dans une sous fonction, cette dernière est placée dans un registre statique, autre abstraction de la pile. Seule elle pourra être stockée dans ce dernier. Ceci permettra à une sous fonction voulant "piocher" cette variable de savoir où regarder. Les variables issues de mère seront référencées par des RegistreMère.
Toutes les variables reçus par référence en paramètre sont référencées par des RegistreIO, et aucune vivacité n'aura été faite dessus. Ce sont des registres "protégés" au même titre que les registres statiques.

2.4.3 Choix

Il a été difficile de trouver cette structure, un tantinet bancale qui plus est. Le maître mot était de séparer le "spim" de l'allocation de registres. Aucune notion de pile ici.

3 Idées générales des différentes passes

3.1 Analyse statique

L'analyse statique est ici un brin conséquente. Elle s'occupe de faire la correspondance nom->registre ou nom->fonction. Elle supprime également tout le sucre syntaxique.

Elle est construite autour d'un "environnement", qui contient toutes les informations utiles ; nom de la fonction courante, fonctions connues, fonctions effectivement construites, entiers connus. A chaque déclaration de sous fonction, il est dupliqué, pour éviter qu'une sous-fonction n'influence une sur-fonction.

Une première étape transforme le code en miniscule, pour une manipulation plus aisée.

Cette analyse statique génère un peu de code intermédiaire, notamment lors d'appels de fonctions ; lors de l'analyse d'une expression, l'analyse statique renvoie non seulement le type de l'expression et sa construction, mais aussi un préembule (une liste d'instructions à exécuter avant l'évaluation de l'expression). Pour les fonctions par exemples, il a été choisi que tous les paramètres soient passés en référence, quitte à faire des copies dans des registres temporaires. Mieux encore, la valeur de retour est passé en référence à la fonction, ce qui permet une homogénéisation du programme : il n'y a plus de distinction procédure fonction, et une fonction renvoyant un tableau n'est plus un problème.
Le fait que l'analyse statique s'occupe de ce type de problème a permis notamment à ce que ce soit la même personne qui s'occupe de la génération du code intermédiaire des fonctions et de la génération spim de ces dernières, afin d'avoir une plus grande cohérence et un nombre de bugs plus faible.

L'utilisation de tableaux constants ou de string déclenche leur enregistrement dans tableauConsEnt et tableauString.
Il y a présence de fonction utiles, qui ont été historiquement codé en pascal, puis incorporé directement (dans globale, cf fin de analyse_statique.ml).
De grandes lourdeurs sont présentes dans cette analyse, notamment pour la construction des tableaux ou des tableaux de constantes, et elle a pris la plus grande partie du temps de Élie ; mais en fin compte, elle est facilement réductible à un simple parcours inductif de la structure.

Se référer à analyse_statique.ml pour de plus amples informations.

3.2 Génération de code intermédiaire

Cette passe consiste en la linéarisation du code, la structure inductive des entiers et des booléens disparaît, et les grosses structures se réduisent à des label, test et jump.
Elle génère une multitude de registres virtuels (non limité). Par exemple, engendrer l'entier Plus(e1,e2) dans le registre virtuel rv1 revient à créer deux registres virtuels rv2 et rv3, à engendrer les entier e1 et e2 dans rv2 et rv3, et faire l'opération rv1 <- rv2+rv3.
Les registres IO sont séparés des autres registres. Si i est l'identifiant d'un registre IO, alors il se retrouve transformé en un RegistreIO(i), afin de faciliter les passes qui suivent.
Juste après cette génération de code intermédiaire, une décoration de chaque fonction est mise en place, qui est la suivante : une liste des variables utilisées par la fonction courante, et une liste des variables utilisées par les fonctions filles (afin de pouvoir générer les registres statiques par la suite).

3.3 Analyse de vivacité et allocation des registres

Le choix ici a été de faire l'analyse de vivacité avant la génération de code, pour éviter de manipuler la pile par ici. Ceci a problablement aidé à cibler l'origine d'erreur de pile.

L'analyse de vivacité constitue en une implémentation de l'algorithme de point fixe.
A chaque fonction dans notre structure de donnée est associée une liste d'instruction correspond au code de cette fonction. L'on crée tout d'abord le graphe de flot de contrôle de ce code, chaque noeud étant du type

*noeud = mutable use : (int*int) list; mutable def : (int*int) list; mutable entrant : (int*int) list; mutable sortant : (int*int) list; mutable succ : int list;;*

Puis l'on a appliqué l'algorithme du point fixe pour avoir les registres vivants en in et out de chaque instruction et ainsi créer le graphe d'interférences.
Afin d'obtenir une allocation des registres, nous avons utilisé la librairie ocamlgraph pour effectuer une coloration du graphe d'interférence. Les couleurs 1 à 16 représentent de véritables registres spims, tandis que les couleurs > 16 correspondent à des registres virtuels et sont donc placés sur la pile.
Avant de générer l'Ast4, une table de hachage est créée, celle-ci a comme clées les identificateurs des fonctions (c'est à dire des listes d'entiers) et renvoie à chaque clé une table de hachage correspondant a une allocation des registres statique (c'est à dire une table de hachage dont les clées sont des int et les éléments des int, par exemple, si (3,1) est un élément de cette table,

alors au registre 3 est associé le registre statique 1). Un registre est associé à un registre statique s'il est utilisé par une fonction fille.
L'étape de génération de l'ast4 a été changé au dernier moment en détresse (mercredi soir), car un bug a été repéré lorsque dans certaines circonstances, nous utilisions une variable mère. Cette dernière n'était sauvegardé qu'aux appels de procédures. Il eut fallu détecter la mort de cette variable pour la sauvegarder à ce moment, et la restituer lors de sa résurrection. Ceci est difficile à détecter avec le système actuel d'allocation de registres ; Pour pallier ce problème, nous mettons à jour les registres statiques dès que le registre correspondant change. Cela implique beaucoup d'accès à la pile, ainsi que de nombreux Move inutiles, d'où une petite étape d'optimisation supplémentaire pour supprimer quelques instructions. Avec plus de temps, ce soin correctif de dernière minute pourrait être rendu plus propre et surtout plus perfomant(car beaucoup de move en l'état). Des bugs restent, comme par exemple si on accède à un registre io d'une mère, le registre statique n'étant jamais utilisé.
Les registres sont tous en cally save, donc tous les registres détruits par une fonction sont sauvés à l'entrée de cette fonction, puis restaurés en sortie.

3.4 Génération code spim

Il faut tout d'abord bien remarquer que toutes les fonctions sont à pile fixe, déterminé à la compilation.
Au cours de l'étape préliminaire de référencement des fonctions dans une table de hachage, il y a justement une décoration des fonctions par des valeurs significatives pour l'allocation de la pile. La place de n'importe quelle variable est parfaitement déterministe et ne dépend absolument pas du chemin d'exécution.

L'enregistrement d'activation est le suivant :

```
#########################################################################
# tableaux                                                              #
# registres_virtuels                                                    #
# registres_statiques (cf Ast4)                                         #
# paramètres passés sur la pile                                         #
# paramètres sauvés à l'appel d'une fonction                            #
# adresse de retour sauvegardé lors d'un appel                          #
# fp de la mère                                                         #
#########################################################################
```

Vu qu'il a une taille prédéterminé à la compilation, il n'y a pas besoin de pointeur de pile. Seul fp est utilisé, libérant ainsi un registre.

La fonction au coeur de la génération est recupere_fp_mere qui s'occupe de remonter les enregistrements jusqu'à trouver l'enregistrement désiré.

Les tableaux sont transmis par adresse lors d'appels de procédures. À ce propos, la fonction resoud_inf s'occupe de déterminer l'ordre adéquat d'affectation de registres IO, pour éviter de supprimer des registres utiles. Il s'agit essentiellement d'une simple recherche de cycles.

3.5 Allocation dynamique

3.5.1 Le tas

La structure du tas est une liste doublement chaînée, chaque bloc possède une entête du type

```
#########################################################################
# Bloc Libre : 0, Bloc utilisé : -1 ,                  ---- 4 octets #
# Pointeur_haut : Si dernier entête, mettre 0          ---- 4 octets #
# Pointeur_bas  : Si premier entête, mettre 0          ---- 4 octets #
#########################################################################
```

3.5.2 Allocation dynamique

Il y a deux fonctions codées à la main en spim :

```
Create : ENTREE, une taille, SORTIE, un pointeur vers le début d'un bloc libre
(créé au besoin) de taille suffisante

Free : ENTREE, un pointeur, SORTIE : Marquage comme étant libre du bloc qui
commence à l'adresse donnée en entrée.

Si le bloc qu'on veut rendre libre est en dessous ou au dessus d'un bloc libre, on
effectue des fusions.
```

4 Ce qu'on aurait eu envie de faire

4.1 Optimisation

Nous avons fait le choix d'exploser la structure lors de la génération du code intermédiaire (avec beaucoup de move) dans l'idée de ne pas bloquer de possibles optimisations par la suite. Cependant, par manque de temps, nous n'avons pu implémenter nos idées.
Par exemple, faire une vivacite un peu plus fine (par exemple, en l'état, les registresvirtuels (ceux de la génération de code intermédiaires) sont colorés de manière "globale" dans le code de chaque fonction, or si un registre est

coloré en 1 à un moment donnée par exemple, nous pouvons néanmoins le colorer d'une autre couleur après sa mort).
Nous aurions aussi voulu faire SSA (Static single assignment form) pour faire de la propagation de constantes aisée et identifier les registres appariés.

Tous les registres sont actuellement cally save.

4.2 Sucre syntaxique

Il manque des structures, des primitives plus riches sur les références, l'utilisation de tableaux comme vrai type primitif (pouvoir faire f(4)[5] par exemple)...

5 Utilisation du compilateur

Après compilation avec le fichier Makefile Makefile.pascal (make -f Makefile.pas), le programme pasc est créé.
Il suffit de l'exécuter avec comme flux d'entrée le fichier source, pour que se trouve dans le flux de sortie le fichier compilé.

6 Exemples

Cf le dossier "exemples", qui comme son nom l'indique, contient quelques programmes révélateurs des performances du compilateur.
Les fonctions numériques marchent, on a préféré mettre des exemples plus intéressants.

6.1 Listes simplement chaînées, et triées !

L'exemple se trouve dans le fichier liste.pas.
Implémentation de listes simplement chaînées, d'un algorithme d'insertion et de suppression en tête, et d'un algorithme d'insertion et de suppression d'élement dans une liste triée. Le programme, interactif, demande la participation de son utilisateur, avec une saisie des élements à insérer ou à supprimer dans une liste triée, et l'affiche !

6.2 Calculatrice

L'exemple se trouve dans le fichier calculatrice.pas.
Il montre comment utilisé un tableau comme valeur de retour de manière (non) astucieuse. Il démontre la flexibilité du compilateur quant à l'utilisation des références et des tableaux. Il montre les dangers des références : une référence pointe sur la zone mémoire du tableau, indépendemment de son

rang de définition. Ce comportement est voulu : une référence est là pour court-circuiter la surcouche tableau.

6.3 Allocation

L'exemple se trouve dans le fichier allocation.pas.
Il montre comment utilisé l'allocation dynamique. Il est suffisemment détaillé lors de son appel pour rendre sa compréhension aisée.

6.4 bcpparam

L'exemple se trouve dans le fichier bcpparam.pas.
Le compilateur est ici torturé, et il y est indiqué des erreurs non encore résolues.