Compilateur

Raphaël Javaux (groupe #14)

Table des matières

1 Langage source

J'ai choisi d'implémenter un **langage à typage statique** reprenant les fonctionnalités de base du langage C.

1.1 Fonctionnalités

- Types de base : **int** et **bool** ainsi que des **tableaux de 1 à N dimensions** de ces premiers ;
- **Qualifieur de type const** pour déclarer des constantes ;
- Déclaration, définition et appel de **fonction** (éventuellement récursif) ;
- **Opérateurs arithmétiques** ($+$, $-$, $*$, $/$, $\%$), **relationnels** ($==$, $!=$, $<$, $>$, $<=$, $>=$) et **logiques** ($\&\&$, $||$) avec respect des règles de priorité et d'associativité ;
- Structures de contrôle **if** et **while** ;
- Insertion de **commentaires** dans le code ;
- **Inférence des types** des variables directement initialisées à l'aide de l'opérateur *auto*.

Le compilateur génère du code LLVM pouvant être, après avoir été compilé vers un code objet binaire, lié à un exécutable écrit en C, par exemple. [1]

1.2 Exemple

QuickSort exprimé à l'aide de ce langage :

```
quicksort(int[] v, int left, int right)
{
    if (left < right) {
        auto i = left + 1;
        auto last = left;
        while (i <= right) {
            if (v[i] < v[left]) {
                last = last + 1;
                swap(v, last, i);
            }
            i = i+1;
        }

        swap(v, left, last);
        quicksort(v, left, last - 1);
        quicksort(v, last + 1, right);
    }
}

swap(int [] v, int i, int j)
{
    auto tmp = v[i];
    v[i] = v[j];
```

1. Le langage ne dispose pas de base de fonctions d'entrées/sorties.

```
    v[j] = tmp;
}
```

2 Implémentation

2.1 Langage d'implémentation

Mon compilateur est écrit en **Haskell** avec l'aide de **Parsec**, une librairie de *combinateurs monadiques* permettant de réaliser facilement des parseurs.

Haskell est un **langage fonctionnel pur**. Ceci signifie que tous les objets du langage sont immuables, c'est à dire que leurs valeurs ne peuvent être modifiées après avoir été assignées.

Haskell est également un langage au **typage statique fort** remarquablement strict. Contrairement à d'autres langages statiques comme Java ou C, tous les types sont déterminés de manière univoque à la compilation (il n'existe pas d'équivalent à *Object* ou *void **). Le vérificateur de typage est fortement similaire à celui d'un outil de vérification de preuves mathématiques comme *Coq*.
Cette caractéristique à l'énorme avantage d'éliminer la majorité des bugs que l'on rencontrerait avec un autre langage lors de l'exécution dès la compilation. Par exemple, le déréférencement d'une référence nulle (ou pointeur nul) ne peut se faire inconsciemment.

La rigueur du système de typage de Haskell s'est avérée extrêmement efficace lors de la réalisation de ce projet : une fois que toutes les erreurs de compilation aient été résolues, le vérificateur sémantique ainsi que le générateur de code se sont exécutés correctement dès le premier lancement.

Les programmes écrits en Haskell utilisent abondamment le concept de **monade**.
Une monade permet de gérer élégamment les changements d'état entre deux instructions. Ce projet en fait largement usage.

L'utilisation de monades pour abstraire les transitions d'état du parseur et du générateur de code a permis de réduire astronomiquement la taille du code source du compilateur. Ainsi, l'entièreté de ce projet comporte moins de **700 lignes de code**, comme présenté sur le tableau suivant :

	blank	comment	code
Définition de l'arbre syntaxique	30	21	81
Préprocesseur	2	3	4
Parseur et vérificateur sémantique	74	72	344
Générateur de code	66	51	240
Autre	2	0	12
SUM	174	147	681

2.2 Structure générale

La compilation d'un fichier source se fait en trois phases :

- Pré-traitement des sources par le **préprocesseur**. Celui-ci accepte un flux de caractères en entrée et retourne un nouveau flux dont les commentaires ont été supprimés ;
- **Parsing et vérification sémantique** des sources. Cette phase accepte un flux de texte en argument (ayant éventuellement passé la phase de précompilation) et retourne soit un arbre syntaxique, soit un message d'erreur ;
- **Génération du code** LLVM IR. Le générateur de code génère le code LLVM sous forme textuelle à partir d'un AST généré par le parseur et le vérificateur sémantique.

2.3 Préprocesseur

Le préprocesseur supprime simplement le contenu des commentaires (qui commencent par le caractère # et qui continuent jusqu'à la fin de la ligne).

2.4 Parseur et vérificateur sémantique

Par simplicité, j'ai choisi d'implémenter le vérificateur sémantique dans le parseur.
Cela rend le parseur un peu moins élégant, mais a néanmoins deux avantages :

- Lorsque que j'émets une exception dans le parseur (mauvais types, variable inexistante ...), je peux facilement **afficher la trace qui indique à quelle ligne le problème s'est produit** ;
- Je n'ai pas besoin d'une **structure de données intermédiaire** entre le parseur et le vérificateur sémantique, les données parsées étant vérifiées dès la fin d'une expression. De plus, une telle structure de données devrait garder les informations de la position (ligne/colonne) de chaque instruction dans le code source de départ si l'on souhaite conserver des messages d'erreur précis.

Le parseur reste néanmoins, à mon opinion, raisonnablement lisible, car très concis.

2.4.1 Parsec

J'ai utilisé une librairie de parseur monadique pour réaliser le parseur. Il ne s'agit pas d'un générateur de parseurs à proprement dit, mais d'un ensemble de primitives simples (appelées **combinateurs**) capables de parser des éléments simples d'un flux de texte. Ceux-ci peuvent **être assemblés** pour former eux-même des combinateurs plus complexes capables de parser des documents plus évolués. Ceci permet d'avoir un contrôle total sur la définition et l'implémentation du parseur. Notamment, par exemple, il n'existe pas de primitive permettant de parser des opérateurs binaires avec associativité, j'ai du donc implémenter cet algorithme à l'aide de combinateurs moins complexes par moi-même.

Un parseur capable de reconnaître la définition du type d'une variable (comme *const int* ou *auto*) peut être définit de cette façon :

```
-- Combinateur retournant la valeur CQualConst s'il est arrivé à parser la
-- chaine "const", CQualFree sinon.
typeQual =     (string "const" >> return CQualConst)
          <|> return CQualFree

-- Combinateur parsant les chaines "int" ou "bool".
typeSpec = string "int" <|> string "bool"

-- Combinateur parsant le type d'une variable.
-- Retourne le qualificateur et le type éventuel (si non-'auto').
varType = do
    qual <- typeQual
    spaces                          -- Ignore les espaces après la qualificateur.
    (    (do t <- typeSpec
             return (qual, Just t))
     <|> (string "auto" >> return (qual, Nothing)))
```

Dans l'exemple précédent *string* et *space* sont des combinateurs primitifs de Parsec. [2]
Il n'y a **pas de transfert explicite de l'état du parseur** (flux de caractères, gestion des erreurs ...) entre deux combinateurs, ceci étant géré implicitement à l'aide d'une monade qui définit la sémantique des opérateurs <|> (alternative), >> (séquencement) ou *do* (séquencement avec variables (définies avec <-)) [3].

2.4.2 Table de symboles

La table de symboles de la librairie standard d'Haskell est implémentée à l'aide d'un **arbre binaire non-mutable**. C'est à dire que l'opération d'ajout d'un élément à la table de symboles retourne un nouvel arbre, sans modifier l'arbre original.
Comme les arbres ne peuvent pas être modifiés, l'ajout d'un élément à un arbre n'implique pas le copie complète de celui-ci, mais uniquement l'allocation d'un certain nombre de nouveaux nœuds (logarithmique par rapport au nombre d'éléments dans la table de symboles) car la quasi totalité de l'arbre original peut être partagée entre les deux arbres.

Cette structure s'avère parfaitement optimale pour gérer la table des symboles contenant les variables. En effet : lorsque le compilateur entre dans une nouvelle *scope*, la copie de l'arbre s'exécute en temps constant (il s'agit de simplement copier le pointeur de la racine de l'arbre) alors que la définition d'une nouvelle variable s'exécute en temps logarithmique. Lors de la sortie d'une *scope*, il suffit

2. *string*, *space*, *typeQual*, *varType* . . .sont toutes des fonctions. Les arguments des fonctions en Haskell ne sont pas entourés de parenthèses.

3. L'utilisation des monades est plus simple dans le générateur de code. L'exemple de la section 2.5 me semble plus accessible pour comprendre l'intérêt et le fonctionnement des monades.

de restaurer la référence vers la racine de l'arbre avant d'être entré dans celle-ci, le garbage collector se chargeant de supprimer les arbres qui ne sont plus utiles.

2.4.3 Gestion des erreurs

Comme indiqué plus haut, en exécutant la vérification sémantique dans la même monade que celle utilisée par Parsec (dont l'état connaît la position du parseur dans le flux de caractères), je suis capable d'**émettre des messages d'erreur indiquant précisément l'endroit où celle-ci s'est produite**.
Par exemple, si la ligne 26 d'un fichier source contient l'instruction suivante :

```
int tmp = v[true];
```

Mon compilateur émettra l'erreur suivante :

```
"stdin" (line 26, column 23):
Subscripts must be scalar integer expressions
```

2.4.4 GADT

J'avais prévu au départ d'utiliser des *Generalized Algebraic Data Types* (une extensions aux types de données d'Haskell) pour définir l'arbre syntaxique généré par le vérificateur sémantique.

Avec un arbre déclaré de cette manière, le vérificateur de type de Haskell aurait été apte à **démontrer la cohérence des fonctions générant cet arbre** par rapport à un ensemble de prédicats.
Par exemple, il aurait été impossible de programmer un vérificateur sémantique produisant un arbre où une constante serait déclarée sans être assignée, où un entier serait assigné à une variable booléenne, où un tableau serait utilisé sans que tous les indices de toutes ses dimensions ne soient renseignés, où une valeur/variable de type booléen serait appliquée à un opérateur arithmétique ...

Cependant, cela rajoute un ordre de complexité à l'écriture du code du vérificateur sémantique à cause d'un accroissement de la rigidité du vérificateur de type de Haskell. J'ai pour cela abandonné cette approche pour mon projet, préférant un arbre moins strict mais plus facile à utiliser.

Cet arbre « typé » est toujours défini dans le fichier *src/Language/Coda/-GAST.hs* du projet, même s'il n'est plus utilisé dans la version finale de mon compilateur.

2.5 Générateur de code

J'ai choisi de générer le code LLVM sous forme textuelle, même s'il existe un binding en Haskell pour l'API d'LLVM. Le générateur de code génère le code LLVM à partir d'une instance de l'AST.

La génération du code LLVM fonctionne sur une combinaison de deux monades. Une première monade *Writer* à la charge d'accumuler les caractères de sortie du compilateur sur un flux textuel et de transmettre ce dernier implicitement entre chaque fonction du générateur alors qu'une seconde monade *State* va maintenir deux compteurs numériques utilisés pour générer, respectivement, les noms des identifiants des registres et ceux des labels des blocs de code.

Ceci permet d'obtenir un code très déclaratif en définissant un petit ensemble de primitives générant du code LLVM et modifiant les différents états des monades. Par exemple, voici le code Haskell commenté générant le code LLVM d'une instruction *while* :

```
-- Génère une instruction while composée d'une garde nommé 'guard' et d'un bloc
-- d'instruction nommé 'stmts'.
cStmt (CWhile guard stmts) = do
    guardLabel <- newLabel -- newLabel alloue un nouveau label.
    loopLabel  <- newLabel -- Cette fonction incrémente le compteur des
    endLabel   <- newLabel -- identifiants des labels de la monade State et
                           -- retourne le nom du nouveau label correspondant.
    branch guardLabel      -- Génère les instructions pour brancher sur le label
                           -- de la garde de l'instruction while sur le flux de
                           -- caractères de la monade Writer.

    tellLabel guardLabel   -- Ajoute sur le flux de caractères l'instruction
                           -- LLVM indiquant le début d'un nouveau bloc nommé
                           -- avec le label précédemment alloué.
    cond <- cExpr guard    -- Génère le code la garde et retourne le nom du
                           -- registre contenant le résultat.
    branchCond cond loopLabel endLabel -- Branche en fonction de la valeur du
                                       -- registre retourné par la génération du
                                       -- code de la condition.

    tellLabel loopLabel
    cCompoundStmt stmts    -- Génère (récusivement) le code des instructions du
                           -- bloc de code de l'instruction while.
    branch guardLabel

    tellLabel endLabel
```

Ici, on peut bien voir que les deux monades permettent de transmettre implicitement le flux de caractères et les compteurs. Par exemple, on pourrait écrire en place quelque chose de similaire à ceci, où chaque fonction du générateur de code recevrait systématiquement les états du flux de caractères et des compteurs, et retournerait systématiquement les nouveaux états de ceux-ci [4] :

```
-- La fonction de génération accepte ici deux arguments supplémentaires pour le
-- flux de caractère et les compteurs.
```

4. On ne peut changer la valeur d'une variable en Haskell, d'où que l'on retourne systématiquement les nouveaux états, même s'ils restent inchangés

```
cStmt stream0 counters0 (CWhile guard stmts) = do
    (guardLabel, stream1, counters1) <- newLabel stream0 counters0
    (loopLabel, stream2, counters2)  <- newLabel stream1 counters1
    (endLabel, stream3, counters3)   <- newLabel stream2 counters2

    (stream4, counters4) <- branch stream3 counters3 guardLabel

    (stream5, counters5) <- tellLabel stream4 counters4 guardLabel
    (cond, stream6, counters6) <- cExpr stream5 counters5 guard
    (stream7, counters7) <- branchCond stream6 counters6 cond loopLabel endLabel

    (stream8, counters8) <- tellLabel stream7 counters7 loopLabel
    (stream9, counters9) <- cCompoundStmt stream8 counters8 stmts
    (stream10, counters10) <- branch stream9 counters9 guardLabel

    (stream11, counters11) <- tellLabel stream10 counters10 endLabel

    -- Retourne les nouveaux états des flux de caractères et des compteurs.
    return (stream11, counters11)
```

C'est exactement ce que fait la monade implicitement. En réalité, on peut voir les fonctions d'une monade comme des fonctions particulières ne retournant pas une valeur mais retournant plutôt une nouvelle fonction qui, en acceptant un ensemble d'états, retourne alors cette valeur avec un nouvel ensemble d'états. Le passage des états à travers les fonctions d'une monade se fait donc, paradoxalement, à l'aide de leurs valeurs de retour. La monade Parsec du parseur utilise le même principe pour transmettre l'état du parseur (position dans le flux de caractères, exceptions ...).

3 Compilation du compilateur

Le compilateur peut être compilé à l'aide du compilateur **GHC**.

Cabal est un utilitaire permettant d'automatiser la compilation et l'installation de programmes écrits en Haskell (similaire aux Ruby Gems).

GHC et cabal [5] sont disponibles dans les dépôts de la plupart des distributions Linux. Pour les autres systèmes d'exploitation, un installateur combinant ces deux logiciels est disponible à cette adresse : http ://www.haskell.org/platform/.

Une fois GHC et cabal installés, il est possible d'installer les dépendances du compilateur (notamment Parsec) en se plaçant à la racine du projet et en exécutant ces deux commandes :

```
cabal update
cabal install --only-dependencies
```

Pour compiler le compilateur avec cabal, exécuter les deux commandes suivantes :

```
cabal configure
cabal build
```

L'exécutable compilé se nomme *coda* et se trouve dans le répertoire *dist/-build/coda/*.

5. Parfois nommé cabal-install

4 Exécution du compilateur

Le compilateur accepte en entrée standard les sources et retourne sur la sortie standard le code LLVM IR correspondant. Par exemple, pour obtenir le code LLVM correspondant au fichier source contenant le Quicksort :

```
./dist/build/coda/coda < examples/quicksort.coda
```

Le code LLVM généré peut alors être passé à *llc* pour générer le code objet binaire à l'aide d'un pipe Unix :

```
./dist/build/coda/coda < examples/quicksort.coda | llc -O2 -filetype=obj > quicksort.o
```

Ce code objet peut ensuite être lié avec un exécutable, écrit par exemple en C.

Le répertoire *examples/* contient une implémentation du QuickSort ainsi qu'une implémentation du jeu de la vie (utilisant des tableaux à deux dimensions) avec des codes C permettant de les utiliser. Un *Makefile* permet de compiler les exécutables **quicksort** et **game_of_life**.