

Compilation de C—: phase d’analyse

Projet « Programmation 1 »

Sur la deuxième partie du premier semestre et le début du second semestre, vous aurez comme objectif de coder un compilateur pour un sous-ensemble du langage C, que nous appellerons le langage C—. Ce compilateur devrait prendre en entrée un fichier contenant du code en C—, contrôler que le code est correct (c’est-à-dire, qu’il respecte les types et les règles statiques du langage), et générer du code LC3¹.

La phase d’analyse doit être codée pendant la deuxième partie du premier semestre. La date limite du *rendu complet* de cette partie est le *20 décembre 2024*. Cette phase se décompose en plusieurs étapes :

- I. L’analyse lexicale et l’analyse syntaxique sont décrites dans ce document et elles sont déjà codées dans les sources qui vous sont distribuées.
- II. Le typage est décrit dans ce document. Le rendu intermédiaire pour cette étape doit être fait *au plus tard le vendredi 22 novembre 2024*.
- III. La création du graphe de flot de contrôle (Control Flow Graph – CFG) et trois analyses utilisant le CFG (décrites dans un prochain document) :
 - a. l’analyse de constantes,
 - b. l’analyse de code « mort » (non accessible),
 - c. l’analyse d’initialisation (une variable est déjà initialisée avant son utilisation).

Le rendu intermédiaire pour cette étape doit être fait *au plus tard le vendredi 13 décembre 2024*.

Les rendus intermédiaires seront utilisés par les encadrants du projet afin de vous faire des retours sur votre projet et d’apprécier l’avancement de votre travail. Le rendu final et ceux intermédiaires seront faits sur la forme d’une archive compressée (.zip ou .tar.gz) d’un répertoire contenant les sous-répertoires suivants :

- **src** : vous y déposerez le code OCaml de votre projet ainsi que les fichiers utilisés pour le compiler,
- **tst** : les programmes écrits en C— (extension de fichier .c) que vous avez utilisés pour tester votre compilateur. Chaque programme doit être accompagné d’un commentaire ou d’un fichier ayant le même nom mais l’extension .log contenant les résultats de la phase d’analyse.
- **doc** : la documentation (en format textuel Markdown ou LaTeX) qui décrit l’architecture de votre code, les choix de codage, les résultats obtenus, les tests utilisés.

L’archive distribuée sur eCampus avec le code de la première étape respecte cette architecture. Il faut la préserver et s’inspirer pour les étapes que vous coderez.

Ce document explique dans une première partie les restrictions de lexique et syntaxe du C—. Puis, il donne les règles de typage à coder pour la seconde étape. Enfin, il décrit les notions (CFG, analyses) pour la troisième étape. La phase de synthèse (génération de code LC-3) sera le sujet des premières séances du projet du module « Langages formels » au second semestre.

1. <http://lc3tutor.org/>

1 Syntaxe

Le C`--` contient des notions déjà vues dans le langage C (variable, fonction, constantes entières, type entier et adresse) mais beaucoup plus simplifiés. Les principales restrictions sont présentées dans cette section.

1.1 Lexique et analyse lexicale

Le langage C`--` ne contient qu'un ensemble restreint de constantes, les constantes entières. Les constantes entières peuvent être écrites en base 10, 8 ou 16. Il résulte que les seuls types utilisables sont le type `int` et les types adresse. Le type adresse peut faire référence à un type `int`, à un type fonction, ou à un autre type adresse. Les déclarations multiples de variables ne sont pas permises ; les variables peuvent être initialisées à la déclaration.

Les opérateurs arithmétiques et de comparaison sont ceux du C. Les opérateurs logiques sont admis, mais pas les opérateurs bit-à-bit. Les expressions peuvent contenir des appels de fonction. L'expression conditionnelle `((a)?b:c)` est aussi autorisée.

Les commentaires et les identificateurs sont ceux du C.

Les instructions sont restreintes à l'affectation, décision (`if`, `if ... else`) et aux boucles (`for` et `while`). Les instructions de saut (`break`, `continue`, `goto`) ne sont pas présentes.

Le lexique du C`--` est formalisé dans le fichier `src/cllexer.mll`. Il est compilé avec `ocamllex` pour générer le code OCaml de l'analyseur. Cette façon de compiler est spécifiée dans le fichier `dune` par : `(ocamllex (modules cllexer))`.

L'analyseur lexical est appelé dans le fichier `main.ml`, qui contient les appels aux phases principales de la compilation.

1.2 Syntaxe et analyse syntaxique

La syntaxe de C`--` est réduite aux constructions du C qui utilisent le lexique restreint présenté dans la section précédente. La seule différence est pour la déclaration d'une variable de type adresse de fonction, dont un exemple (correct) est donné dans le fichier `tst/test04.c`.

L'analyseur syntaxique du C`--` est codé dans le fichier `src/cparser.mly`, compilé avec `menhir` pour générer le code OCaml de l'analyseur. Cette façon de compiler est spécifiée dans le fichier `dune` par : `(menhir (modules cparser))`.

L'analyseur syntaxique construit un arbre de syntaxe abstraite (AST), valeur de type `file` dans le fichier `src/cast.mli`.

L'analyseur syntaxique est appelé dans le fichier `main.ml`.

Le code distribué permet à présent de transformer des fichiers source écrits en C`--` vers des fichiers en format DOT contenant l'arbre de syntaxe abstraite. Pour cela, il faut :

1. nettoyer les fichiers générés avec : `dune clean`
2. compiler les sources avec : `dune build`
3. appeler l'exécutable généré avec : `dune exec -- ./main.exe --debug test.c`

Ces commandes sont en partie présentes dans le fichier `Makefile` du sous-répertoire `src`.

Affichage de l'arbre syntaxique. Afin de vous aider à lire les AST, nous fournissons un module `src/pretty.ml` qui offre une fonction pour traduire un AST vers le format DOT dans un fichier `.dot`. L'AST en format DOT peut être ensuite visualisée avec l'outil `dot` du paquetage `graphviz` par `dot -v -Tpng -O test_ast.dot`

Attention, vous devez modifier uniquement les fichiers qu'on vous demande à modifier dans le fichier `README.md`.

2 Typage statique

Les règles de typage informelles sont :

1. Les types autorisés sont : entier et pointeur vers un type (par exemple, `int*`, `int**`) qui peut être un type fonction (par exemple `int(*) (int, int)`).
2. Une variable globale est déclarée une seule fois.
3. Une fonction est définie une seule fois.
4. La définition d'une fonction ne doit pas avoir deux paramètres avec le même nom.
5. Une variable locale peut être redéfinie dans un bloc imbriqué.
6. Une variable locale peut avoir le nom d'une fonction, mais alors elle cache la définition de la fonction.
7. Les valeurs affectées doivent être de type de la valeur gauche de l'affectation.
8. L'opérateur « adresse de » (&) peut être appliquée que sur une variable qui a un type entier ou adresse d'un entier.
9. La conversion implicite entre type adresse et type entier n'est pas permise.
10. Entre les valeurs entières, les opérations arithmétiques présentes dans la syntaxe sont autorisées.
11. Entre les valeurs de type adresse du même type qui n'est pas un type fonction, la seule opération permise est la soustraction².
12. Entre une valeur de type adresse d'un type qui n'est pas fonction et une valeur entière, les opérations permises sont : l'addition et la soustraction.
13. Une fonction doit être appelée avec le type et le nombre des paramètres correspondant à sa définition.
14. Le bloc d'une fonction doit contenir une instruction `return` avec une expression du type de retour déclarée pour la fonction, dans toutes les branchements du code.
15. Deux fonctions `print_int` et `print_string` sont disponibles pour imprimer un entier respectivement un texte.

Ces règles doivent être codées dans le fichier `ctyping.ml`, dans une fonction `check_file`. Cette fonction renvoie la liste des variables globales et fonctions déclarées, chacune avec leur type si le code analysé est bien typé et lève une exception sinon ; elle signale les erreurs de typage par des messages faciles à interpréter.

2. Ceci est une simplification par rapport à la norme C qui demande que les pointeurs soient dans le même bloc mémoire.

3 Analyses


Le but de cette partie du projet est de préparer la phase de synthèse de code LC-3 en améliorant le code écrit. Les améliorations visées permettent d'éliminer des calculs à l'exécution en les effectuant à la compilation, d'éviter de générer du code qui n'est pas accessible à partir du programme principal et de signaler si une variable est utilisée avant d'être initialisée. Pour effectuer ces améliorations, nous utilisons une représentation spéciale du programme, le graphe de flot de contrôle, décrite dans la section suivante.

3.1 Graphe de flot de contrôle

Un graphe de flot de contrôle (*control flow graph*, CFG) est une représentation d'un programme qui permet de manipuler les chemins d'exécution du programme. Il s'agit d'un graphe dirigé dont les sommets représentent des blocs (séquences) d'instructions simples (affectations, appels de fonctions et évaluation d'expressions) qui se terminent par un saut (sortie de fonction – `return`, sauts conditionnels ou non conditionnels) et dont les arcs indiquent les cibles des sauts.

Plus précisément, le graphe du programme contient un sous-graphe par fonction du programme. Chaque sous-graphe a un sommet d'entrée représentant le début de la fonction et au moins un sommet de sortie qui contient l'instruction `return`. Le sommet d'entrée du sous-graphe de la fonction `main` est le point d'entrée du programme.

Cette représentation du CFG est codée dans les fichiers `cfg.ml*` qui vous sont distribués et doivent être ajoutés à votre code source. Les commentaires dans ce code vous expliquent l'organisation des blocs, comment sont représentées les variables globales, locales ou paramètres des fonctions, la notion de registre (variable temporaire créée pour traduire les instructions en instructions simples). Le choix des instructions simples a été fait pour vous faciliter la phase de synthèse vers l'assembleur.

 : Codez la construction du CFG d'un programme à partir de son AST dans le fichier `ast2cfg.ml` en complétant le code qui vous est donné.

3.2 Propagation de constantes

Cette analyse a comme but de déplacer certains calculs de la phase d'exécution vers la phase de compilation. On parle aussi de simplification de sous-expressions constantes ou propagation de constantes (*constant folding*, CF).

Par exemple, le programme `x=0; if(x) y=y-1; else y=x+1;` peut être traduit directement en `x=0; y=x+1;` donc dans un seul bloc (sommet de CFG) au lieu de trois blocs (3 sommets et deux arcs). La valeur constante de `x` a été propagée dans l'évaluation de la condition ; par conséquent, l'instruction de saut conditionnel peut être supprimée avec l'arc pour la valeur non nulle et le bloc cible de l'arc pour la valeur nulle de la condition a été ajouté au bloc source. En propageant encore la valeur constante de `x` dans ce nouveau bloc, nous obtenons le bloc `x=0; y=1;`

Afin de pouvoir réaliser ces simplifications de CFG, nous avons besoin de calculer, pour chaque instruction simple des blocs l'information suivante : quelle est la valeur de chaque variable quand l'exécution du programme arrive à cette instruction, pour toutes les exécutions possibles. Attention, la notion de variable inclut maintenant les variables globales, locales, les paramètres de fonction et les registres (variables temporaires) introduits pour la traduction de l'AST vers le CFG. L'analyse concernera que les variables de type numérique, car les variables de type adresse n'ont pas beaucoup de valeurs constantes.

En général, une variable a des valeurs différentes pour chaque exécution, quand celle-ci arrive à une instruction simple. L'analyse CF détecte les cas où la valeur est la même chaque fois

qu'une exécution arrive à cette instruction. Un corollaire de cette analyse est la détection de code inaccessible, comme l'instruction `y=y-1` dans l'exemple ci-dessus.

Pour cela, l'analyse associe à chaque instruction simple du programme une information sur les (nouvelles) variables du programme. Comme une instruction simple est identifiée par le bloc d'appartenance et sa position dans le bloc, l'information est une association $(labelBlock, positionInstr) \mapsto (variable \mapsto v)$. Toutefois, cette information n'est pas calculée initialement, donc on aura une association $(labelBlock, positionInstr) \mapsto \perp$ qui nous indiquera également que l'analyse CF n'a pas atteint cette instruction. L'association peut être codée à l'aide de table de hachage ou autres structures de données.

Ensuite, analyse commence une *interprétation abstraite* du programme, en partant du bloc de debut de la fonction `main` et en effectuant un parcours du CFG. Ce parcours est répété tant que l'information associée à une instruction simple a été actualisée par le parcours précédent.

Dans cette interprétation, l'analyse calcule la *liaison abstraite* D qui associe à chaque variable x une *valeur abstraite* v ($variable \mapsto v$ ci-dessus). La valeur abstraite v est soit une constante (entière), soit \top pour indiquer que l'analyse ne peut pas calculer une seule valeur pour x . Ainsi, D est soit \perp (cas initial), soit une fonction partielle (car associe que les variables visibles par l'instruction) :

$$D \in \mathbb{D} \triangleq (Vars \twoheadrightarrow \mathbf{int} \cup \{\top\}) \cup \{\perp\}$$

avec \mathbf{int} l'ensemble de valeurs de type entier.

Si on fixe l'ordre partiel $k \preceq \top$ pour tout $k \in \mathbf{int}$, alors on munit le domaine \mathbb{D} d'un ordre total \sqsubseteq défini par :

$$D_1 \sqsubseteq D_2 \quad \text{iff} \quad \perp = D_1 \text{ ou } D_1(x) \preceq D_2(x)$$

On peut montrer que $(\mathbb{D}, \sqsubseteq)$ forme un treillis complet.

L'interprétation abstraite calcule un plus petit point fixe pour l'association (instruction simple, liaison abstraite) en partant de l'association initiale à \perp et en utilisant le parcours de CFG décrit ci-dessus. Le calcul commence par le point d'entrée du programme qui change la liaison abstraite de la première instruction en utilisant la valeur abstraite (constante ou \top) des variables globales (l'initialisation des variable locales est faite dans les instructions qui suivent).

Lors du parcours de CFG, l'analyse met à jour les liaisons abstraites en appliquant chaque instruction e à la liaison qui lui est associée. Ainsi, l'interprétation abstraite d'une instruction e est définie par :

$$\llbracket e \rrbracket : (\mathbb{D} \setminus \{\perp\}) \rightarrow (\mathbf{int} \cup \{\top\})$$

Par exemple, pour $D = \{x \mapsto 2, y \mapsto \top\}$:

$$\begin{aligned} \llbracket x + 7 \rrbracket(D) &= \llbracket x \rrbracket(D) + \llbracket 7 \rrbracket(D) = 2 + 7 = 9 \\ \llbracket x + y \rrbracket(D) &= \llbracket x \rrbracket(D) + \llbracket y \rrbracket(D) = 2 + \top = \top \end{aligned}$$

Quand l'analyse calcule une liaison abstraite D_2 pour une instruction dont un liaison $D_1 \neq \perp$ est déjà présente, alors la nouvelle liaison D est définie par :


$$D(x) = v \text{ tel que } D_1(x) \preceq v \text{ et } D_2(x) \preceq v$$

Si $D_1 = \perp$, alors D est D_2 .

Quand l'analyse calcule D_2 pour le debut d'une boucle (l'instruction a déjà été visitée dans le parcours courant du CFG), nous avons en général une liaison D_1 associée à cette instruction avec $D_1 \neq \perp$. En utilisant le calcul ci-dessus de D , on peut obtenir $D = D_1$ (donc que $D_2 \sqsubseteq D_1$) ; si c'est le cas, l'analyse n'explore plus le chemin déjà exploré et passe à un autre chemin de la boucle. Sinon, le chemin est exploré encore une fois. Comme le treillis $(\mathbb{D}, \sqsubseteq)$ est de profondeur finie,

on tombera forcément sur le premier cas, donc l'exploration du chemin se termine forcément. Si tous les chemins de la boucle ont été explorés et que la liaison associée au début de la boucle D_1 n'a pas évolué, alors la suite du bloc n'est plus explorée car on a atteint un point fixe.

L'analyse s'arrête quand l'exploration du CFG n'apporte plus de changements aux liaisons abstraites des instructions simples.

 : Codez cette analyse dans le fichier `analysis_CF.ml`.


L'analyse doit renvoyer l'association $(labelBlock, positionInstr) \mapsto (variable \mapsto v)$ calculée par l'analyse CF. Quand l'option `debug` est utilisée, cette association sera affichée comme une liste de lignes, chaque ligne affichant l'instruction simple du CFG (l'étiquette du bloc et la position de l'instruction sont optionnels mais l'instruction peut être utile) et la liaison abstraite qui lui est associée par l'analyse. Pour l'exemple du code `x=0; if(x) y=y-1; else y=x+1;`, l'affichage pourrait être (le code pour la fonction principale et le nom des étiquettes sont simplifiés) :

```
(b0,0): {x:top,y:top} x=1
(b0,1): {x:1,y:top} x
(b0,2): {x:1,y:top} jmpc 11, 12
(l1,0): bot y=y+1
(l1,1): bot jmp f0
(l2,0): {x:1,y:top} y=x+1
(l2,1): bot jmp f0
(f0,0): {x:1,y:2} nop
```

3.3 Détection de code inaccessible

Un corollaire de l'analyse CF est que les instructions pour lesquelles la liaison abstraite est \perp sont inaccessibles dans le programme. On peut donc transformer le CFG pour éliminer ces instructions, les variables utilisées uniquement par ces instructions et les blocs vides (sans instructions simples ou uniquement avec des instructions `nop`).

Attention, une analyse complète de code mort élimine aussi les instructions simples et les variables qui ne sont pas utiles aux calculs qui suivent ces instructions. Par exemple, la traduction instructions simples de l'instruction `x=2+2` serait `x1=2; x=x1+2` (avec `x1` un registre). L'analyse CF donne $x1 \mapsto 2; x \mapsto 4$ après la dernière instruction. Si le registre `x1` n'est plus utilisé après ce code, on peut réduire cette séquence à `x=4`. On ne vous demande de coder cette version complète de la transformation du CFG.


 : Codez dans le fichier `cfg_filter.ml` la transformation de CFG qui utilise le résultat de l'analyse CF pour éliminer les blocs (et les arcs) inaccessibles.

3.4 Analyse d'initialisation

L'analyse d'initialisation de variables détecte l'utilisation de variables non initialisées. Pour cela, elle procède comme l'analyse CF mais avec une liaison abstraite simplifiée à deux valeurs abstraites initialisée – `def`, non-initialisé – `undef` et `def` \preceq `undef` :

$$D \in \mathbb{D} \triangleq (Vars \rightarrow \{\text{def}, \text{undef}\}) \cup \{\perp\}$$

La liaison abstraite associée à la première instruction d'un sous-graphe considère que les variables globales et les paramètres de fonction sont initialisés mais que les variables locales (et les registres) ne le sont pas. L'interprétation des instructions e est modifiée pour cette analyse car l'utilisation d'une variable liée à `undef` donne comme résultat `undef` et affiche une alarme (message) signalant l'utilisation d'une variable non initialisée.

 : Codez cette analyse dans le fichier `analysis_init.ml`.