Design review - Itération 1

Zohour Abouakil
Sofia Boutahar
David Courtinot
Xiaowen Ji
Fabien Sauce

Compte-rendu des décisions prises lors des séances de conception du 26/01 au 28/01

Sommaire

Partie I

Représentation AST et CFG

Après avoir étudié l'API de Clang, nous sommes arrivés à la conclusion que l'AST comportait bien plus de nuances que ce dont nous avions besoin pour construire le CFG. En effet, l'atome pour un CFG est ce qu'on appelle généralement une *instruction* tandis que l'instruction la plus simple donne une représentation AST composée de plusieurs noeuds. Nous avons par ailleurs jugé délicat de gérer à la fois le parsing et la liaison des noeuds du graphe. C'est pourquoi nous avons pris le parti de transformer l'AST en une suite d'objets de plus haut niveau que ses noeuds originaux, qui donneront lieux soit à des blocs du CFG, soit à des macro-blocs contenant un sous-graphe.

I.1 Représentation intermédiaire de l'AST de Clang

I.1.1 Partie Decl

I.1.2 Partie Stmt

Pour cette première itération, on ignore totalement la partie objet de C++ pour se concentrer uniquement sur la partie impérative. En s'inspirant de l'API de Clang, nous sommes arrivés au diagramme de classes suivant :

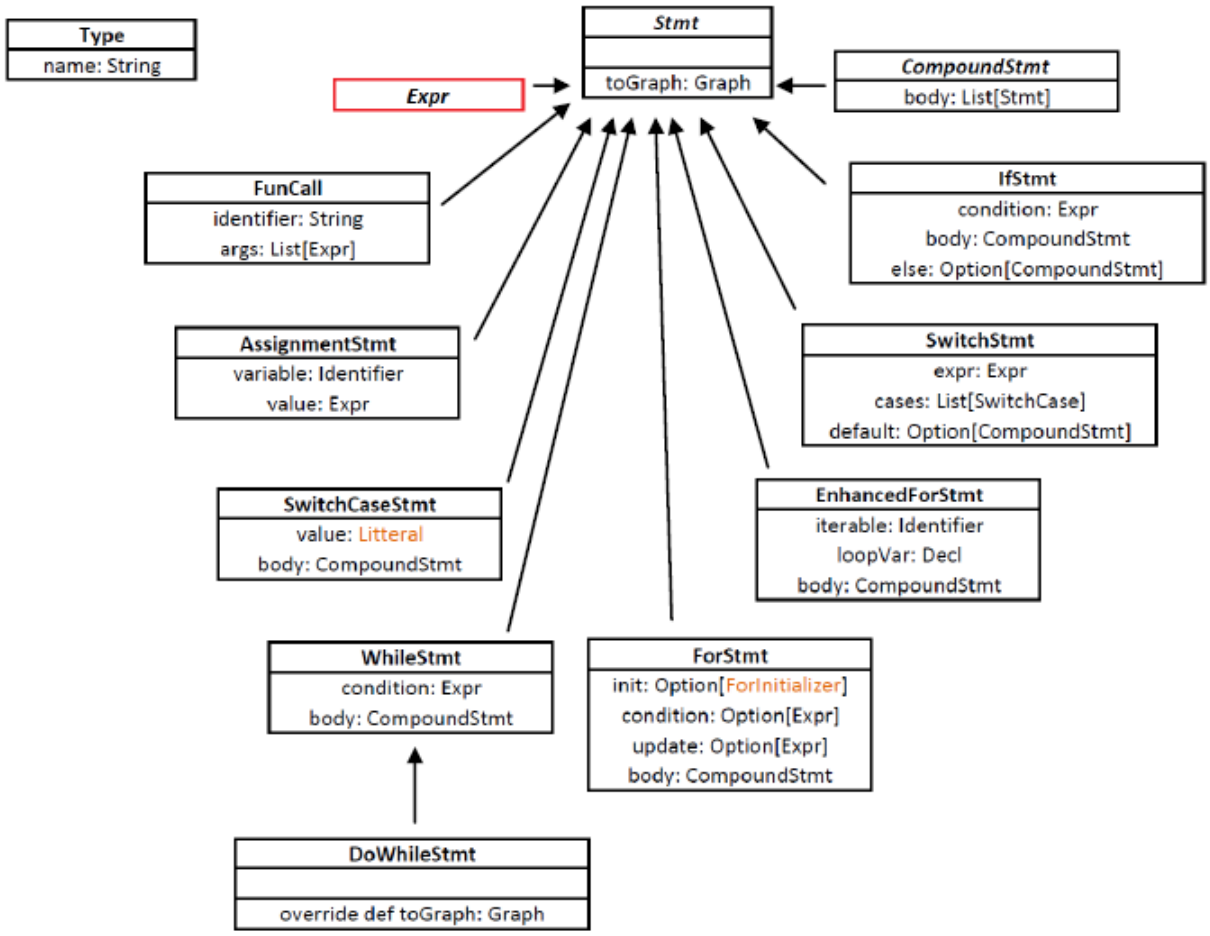

Figure I.1 - Diagramme de classes de la partie (impérative) de **Stmt**

Il est à noter que notre modélisation ne représente pas rigoureusement un code C++. Par exemple, nous n'empêchons pas un *if* de contenir d'instruction *break* (parmi d'autres imprécisions...). Nous avons estimé que ce genre de raffinement compliquerait inutilement la tâche sans rien apporter de plus à l'analyse du CFG. Dans la mesure où le code est déjà vérifié sémantiquement par le compilateur Clang et au vu de nos besoins ultérieurs, il apparaît plus sage de tendre vers un modèle simple. Par ailleurs, certaines classes n'ont pas été clairement définies sur le diagramme précédent, car nous avons volontairement choisi de les représenter à part sur le diagramme ci-dessous. Enfin, d'autres classes n'ont pas été représentées (**BreakStmt**, **ContinueStmt** par soucis de lisibilité).

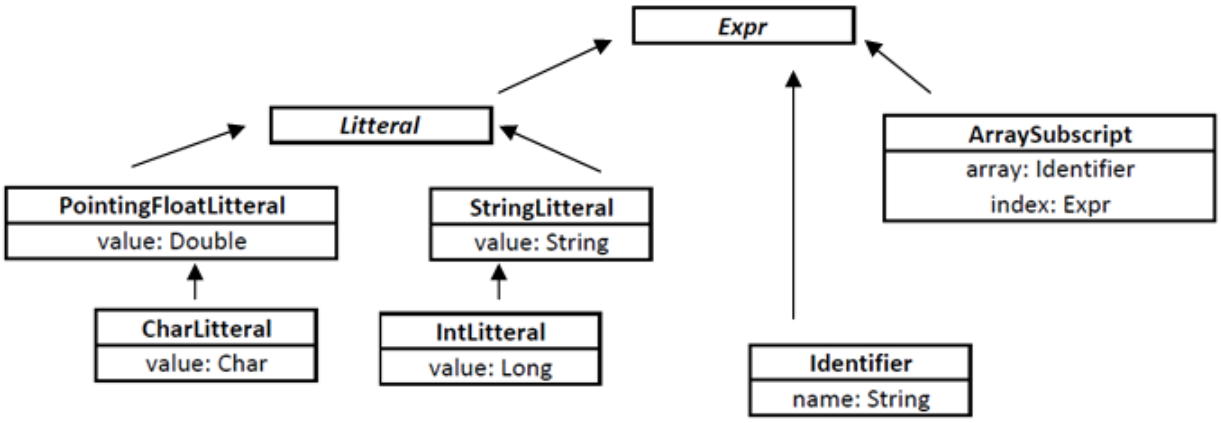

Figure I.2 - Diagramme de classe partiel de la partie **Expr**

Pour finir, nous présentons un diagramme plus complet pour **Expr** mais que nous ne sommes pas certains de conserver, sa complexité étant jugée superflue pour la nature des algorithmes que nous mettront en place sur ces structures.

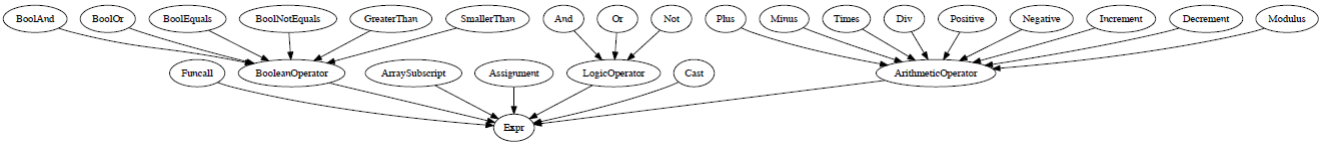

Figure I.3 - Diagramme de classe plus fourni (mais toujours incomplet) de la partie **Expr**

Remarques importantes :

- pour représenter fidèlement le CFG de nos programmes, nous devront tenir compte du mécanisme de fail-fast dans l'évaluation des conjonctions/disjonctions. L'importance de ce mécanisme pour notre projet est illustrée dans la figure suivante.
- toutefois, l'ordre d'évaluation des expressions n'étant pas spécifié en C++ (contrairement à Java qui évalue de gauche à droite), nous limiterons ces considérations à des expressions simples. Nous nous désintéresserons donc aussi des expressions (booléenes ou non) contenant des sous-expressions à effet de bord (incrémentation, assignations...).

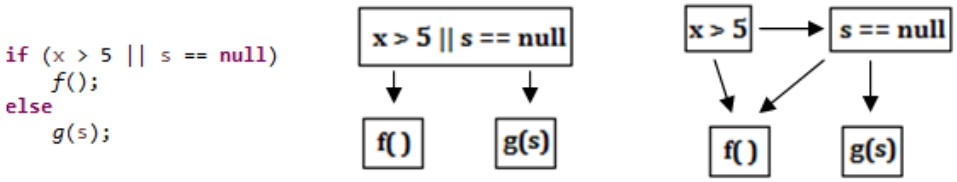

Figure I.4 - À droite, le code considéré. Au centre, le CFG correspondant à la non prise en compte du fail-fast et enfin à droite, prise en compte du fail-fast

Comme on peut le constater sur la figure ci-dessus, si l'on cherche à assurer la propriété *« s est toujours initalisée lorsque g(s) est appelée »*, le premier CFG ne nous permettra pas de conclure (ou bien il concluera **true**, à tort) tandis que le second nous permet de dire qu'il existe des exécutions, selon la valeur de a, pour lesquelles g est appelée avec un paramètre non initialisé (si l'on admet que le fils gauche représente toujours un test réussi et que le fils droit représente un test échoué).

I.2 AST to CFG

Partie II

Model checking