Rapport

Nathan Caracciolo, Arnaud Bihan

3 mars 2022

1 Les origines de LLVM

A l'origine de la création de LLVM en 2003, il devait poser les bases pour un nouveau compilateur capable de dispenser des machines virtuelles similaires à la machine virtuelle Java pour un ensemble de langage d'entrée variable.

Des développeurs Apple se rattachent rapidement au projet afin de créer Clang un compilateur C alternatif à GCC. En effet le compilateur GCC posait de nombreux problèmes aux développeurs d'Apple mais aussi aux développeurs de LLVM, la taille et la complexité du projet commençaient à compliquer son développement. Très rapidement le projet change d'objectif sans changer de nom et propose une pipeline de compilation complète vers du code machine.

Le compilateur LLVM veut servir de base au développement des futurs langages en proposant une pipeline modulable dont le point d'entrée est un langage intermédiaire. La pipeline propose un ensemble d'outils et d'optimisation très performantes. Les seuls éléments restants et nécessaire à la création d'un nouveau langage sont le Parsing et la génération du code intermédiaire.

2 La Pipeline LLVM

La pipeline LLVM est constituée de trois blocs indépendants qui peuvent être changés en fonction des besoins afin de changer le langage source, le langage de sorti de la pipeline ainsi que le nombre d'optimisations qui seront effectuées. Les trois phases de la pipeline LLVM sont le Front-end, la représentation intermédiaire LLVM et le Back-end.

2.1 Front-End

Le front end est la partie modulaire de la pipeline LLVM, il s'adapte au langage source que l'on cherche à compiler. Il existe un grand nombre de front-end permettant de prendre en charge un grand nombre de langage de programmation. Parmi ces langages on peut trouver C, C++, Haskell, Julia, Objective-C, Rust, Swift et bien d'autres. Il s'agit de la seule phase qui est spécifique au langage source et certaines optimisations ou vérifications peuvent y être faite. C'est par exemple le cas du langage Rust qui effectue des vérifications pour sa gestion spécifique de la mémoire lors de cette phase.

2.2 LLVM IR

LLVM IR (ou LLVM intermediate representation) est la représentation intermédiaire utilisée par LLVM, c'est dans ce langage que sont traduit les langages d'entrées par les différents front-end. Il s'agit d'un ensemble d'instructions indépendantes de tout langage, de tout paradigme et de tout système.

Le langage intermédiaire est proche des codes à trois adresses et est souvent décrit comme un assembleur typé. La représentation à trois adresses n'utilise que des calculs ayant au plus 3 opérandes, généralement une combinaison d'affectation et d'un opérateur binaire, comme par exemple :

```
x := y + z
```

Les différentes optimisations proposées par LLVM vont se faire sur ce langage on aura donc des transformations du langage intermédiaire vers ce même langage intermédiaire afin d'enchaîner plusieurs optimisations.

2.2.1 Les optimisations

Comme dit précedemment LLVM effectuent ses optimisations sur son langage intermédiaire et découpe cela en trois phases :

- La première phase est la phase analyse et récupère toutes sortes d'informations utiles sur le programme pour les optimisations mais aussi pour le debogage.
- La deuxième phase est la phase de transformation, elle applique les différentes optimisations grâce aux informations récoltées par la phase d'analyse et transforme le programme au fur et à mesure.
- Et la dernière phase est une phase utilitaire qui regroupe le reste des informations qui ne rentrent pas dans les deux premières catégories.

2.3 Back-end

La pipeline LLVM propose un grand nombre de format de sorti pour s'adapter à tous les supports. Différents assembleur sont pris en compte dont bien sur l'assembleur x86-64. Mais on trouve des formats plus exotiques comme l'assembleur Web permettant de faire tourner des programme dans un navigateur de recherche ou des formats de calculs GPU pour applications parallèlisables.

3 Les front-end LLVM

3.1 Clang

3.1.1 Les origines de Clang

L'idée du compilateur Clang apparait dans les années 2005 avec le besoin croissant d'Apple de trouver une alternative au compilateur phare GCC. En effet le compilateur GCC posaient différent problème à Apple. Premièrement la prise en compte du langage Objective-C (très utilisé chez apple) par le front-end de GCC étaient jugé insuffisant par Apple. Deuxièmement le compilateur de GNU ne s'intégrait pas assez bien dans les environnements de développement d'Apple. Et enfin le fait que GCC soit sous licence GPL, forçant de fait Apple à publier toute modification du compilateur GCC est une raison du développement de ce nouveau compilateur.

Apple décide donc de développer son nouveau compilateur Clang qui respecte des règles plus stricts sur le langage C (le C99) et qui compile aussi le C++ et le langage Objective-C. Ce projet est open-source depuis 2007.

3.1.2 Clang

Le compilateur Clang compile actuellement 96% des paquets Debian. Ce n'est pas un défaut du compilateur Clang mais une conséquence de la trop grande permissivité de GCC dans ses analyses. Par exemple le compilateur gcc compile le code ci-dessous mais pas le compilateur clang :

```
int test() {
    return;
}
```

Idem pour le code suivant :

```
test() {
    return 2;
}
```

Mais on peut noter une très nette amélioration et correction des paquets Debian qui n'était que 85% à compiler en 2011.

3.2 GHC

3.2.1 Les origines de GHC

GHC (pour Glasgow Haskell Compiler) est un compilateur open source pour le langage fonctionnel Haskell. Développé à la base par l'Université de Glasgow. Avec une première version sortie en 1989, c'est le compilateur le plus utilisé qui prend le langage Haskell en entré et génère du code qui pourra être exécuté par la machine compilant.

3.2.2 GHC

GHC fait passer le langage Haskell par des langages intermédiaires. Le premier langage est le STG, c'est un langage fonctionnel SSA-form soit Static Single Assignement. Cela veut dire que ce code :

```
x := 8
x := 4
y := x + 3
```

sera transformé en :

```
x1 := 8
x2 := 4
y1 := x2 + 3
```

Sous cette forme on se rend bien compte que x1 n'est pas utile et pourrait être supprimé par une optimisation. Le langage passe par la suite par C- - un sous-ensemble de C à l'inverse de C++. Le back-end de GHC sert donc à prendre ce code C- - et le compiler vers du code machine. GHC a de base un générateur de code natif, mais avec l'arrivée de LLVM, GHC développe un back-end vers LLVM-IR afin de bénéficier de ce qui a déjà été fait par LLVM. Selon la documentation de GHC, le back-end LLVM offre dans certains cas un temps d'exécution un peu plus rapide que le générateur de code natif notamment sur les arrays, les numerics et sur du code utilisant les vecteurs.

Mais il y a deux problèmes principaux du back-end LLVM :

- il faut avoir LLVM d'installé qui est un effort supplémentaire pour un petit gain de temps par rapport au générateur de code natif
- le temps de compilation en utilisant LLVM est significativement supérieur que lorsque l'on utilise le générateur de code natif

Selon un benchmark, LLVM offre en moyenne un temps d'exécution 4.5% plus rapide que le générateur de code natif mais un temps de compilation 22.7% plus long que le code natif.

3.3 Gallium3D

3.3.1 Les origines de Gallium3D

Gallium3D est un logiciel libre développé en partie par Tungsten Graphics(maintenant VMware) à partir de 2007. Gallium3D ayant pour but de remplacer d'anciens pilotes graphiques de la librairie Mesa3D afin de simplifier la prise en compte de nouvelles versions d'OpenGL. OpenGL étant un ensemble de fonctions normalisés pour faire du calcul d'image 2D et 3D.

Mesa3D est une bibliothèque graphique libre commencé en 1993 par Brian Paul et qui fourni une implémentation d'OpenGL libre de droit. Elle est utilisé sous plusieurs plateformes dont Linux. L'accélérateur de rendu purement logiciel LLVMPIPE du projet Gallium3D est utilisé par défaut par Mesa3D depuis 2014.

Gallium3D utilise une représentation intermédiaire pendant sa transformation de programme : la représentation TGSI (Tungsten Graphics Shader Infrastructure) avant d'utiliser la pipeline LLVM et ses optimisations.

3.4 Swift

3.4.1 Les origines de Swift

Swift est un langage de programmation créé pour développer des applications dans l'environnement d'Apple (IOS,Mac, AppleTV,...). Le projet a commencé en 2010 par Chris Lattner pour finalement publier une première version en 2014. Le projet est passé Open-Source en 2015 et depuis beaucoup de gens (principalement Apple) participent au projet. Swift a été motivé pour faciliter la programmation dans l'environnement Apple en créant un langage plus concis et simple que l'Objective-C. De ce fait, afin de ne pas pousser les développeurs à refaire tous les logiciels écrit en Objective-C, Swift a été pensé pour bien se coupler avec les vieux codes d'Apples écrit en Objective-C. Apple étant à l'initiative du projet LLVM/Clang, ils vont utiliser LLVM pour compiler leur nouveau langage.

3.4.2 Swift

Apple va créer un langage intermédiaire appelé Swift Intermediate Language (SIL). Ce dernier va permettre au compilateur Swift de faire certaines optimisations qui seraient à un niveau trop haut d'abstraction pour le code transformé en LLVM-IR. La transformation directe en LLVM-IR ferait perdre trop d'informations sur le code. Le compilateur Swift effectue donc une série d'optimisations sur le langage SIL avant de traduire ce code en LLVM-IR puis laisser LLVM finir le travail.

4 Références :

— https ://en.wikipedia.org/wiki/LLVM
— https ://llvm.org/docs/
— https ://fr.wikipedia.org/wiki/Gallium3D
— https ://fr.wikipedia.org/wiki/Mesa_3D
— https ://fr.wikipedia.org/wiki/LLVMpipe
— https ://en.wikipedia.org/wiki/Clang
— https ://clang.debian.net/
— https ://downloads.haskell.org/ghc/latest/docs/html/users_guide/
— https ://fr.wikipedia.org/wiki/Glasgow_Haskell_Compiler
— https ://gitlab.haskell.org/ghc/ghc/-/wikis/
— https ://andreaspk.github.io/posts/2019-08-25-Opinion%20piece%20on%20GHC%20backends.html
— https ://www.swift.org/swift-compiler/*
— https ://fr.wikipedia.org/wiki/Swift_(langage_d%27Apple)
— https ://www.youtube.com/watch ?v=IR_L1xf4PrU