LLVM – EINE (KURZE) EINFÜHRUNG

… oder wie ich lernte einen Compiler zu bauen!

Markus Döring - Steffen Weber - Marcel Wiaterek - Alexander Zautke

Inhaltsverzeichnis

LLVM – Eine (kurze) Einführung

Für die Projektarbeit im Rahmen der Wahlpflichtveranstaltung „Compiler und Interpreter“ bei Prof. Erich Ehses hat sich unser Projektteam für das Thema LLVM entschieden. Ziel war es, ein grundlegendes Verständnis für die Arbeits- und Funktionsweise von LLVM zu erlangen und dieses Wissen in einem eigenen, praktischen Projekt umzusetzen.

In diesem Dokument wird LLVM und das hierauf aufbauende Projekt „subset“ vorgestellt. Dazu wird zuerst auf die Entstehungsgeschichte eingegangen, anschließend wird der Aufbau und die Architektur von LLVM skizziert und schließlich die LLVM-eigene Sprache LLVM IR vorgestellt. Im Anschluss wird das Abschlussprojekt „subset“ präsentiert, für welches eine eigene, kleine Programmiersprache entworfen wurde, die mit Hilfe von LLVM kompiliert und optimiert werden kann. Die Sprache heißt „subset“, da es sich um eine Teilmenge der Funktionen der Sprache C handelt.

Entstehungsgeschichte

Bevor auf die Entstehungsgeschichte von LLVM eingegangen wird, wird zunächst ein kurzer Überblick auf die Compilerlandschaft im Jahre 2000, also vor der Entwicklung von LLVM gegeben.

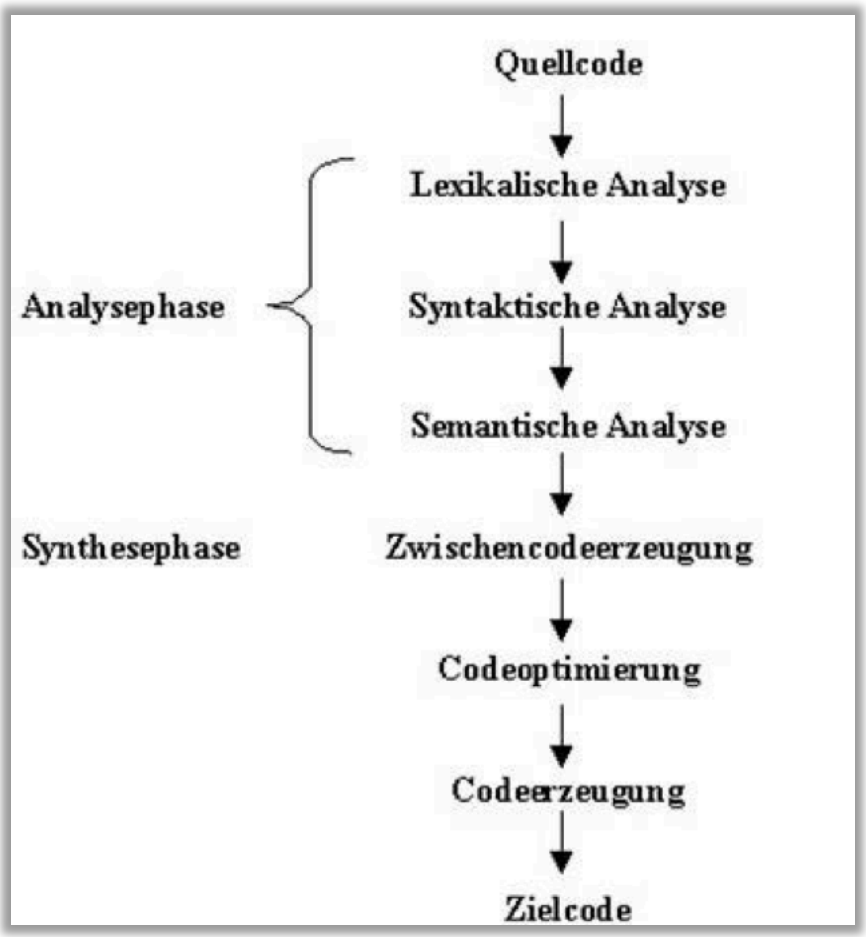

Abb. 1: Aufbau eines Compilers

Betrachtet man den Ablauf bzw. Aufbau eines klassischen Compilers (Abb. 1), könnte der Gedanke entstehen, dass dieser aus einzelnen Modulen besteht, welche beliebig ausgetauscht werden können. Hieraus folgt, dass z.B. eigentlich die semantische Analyse unabhängig von der lexikalischen Analyse sein sollte. Da jeder Compiler einen ähnlichen Aufbau besitzt, sollte es deshalb auch möglich sein, Komponenten zwischen den Compilern zu tauschen, um Code-Doppelungen zu sparen.

Obwohl sich die einzelnen Compiler im Grunde den gleichen Aufbau und die gleiche Funktionsweise teilten, waren sie im Jahr 2000 Tools, welche nur für einen speziellen Anwendungszweck bzw. eine bestimmte Sprache entwickelt wurden. Es gab kaum eine Möglichkeit einen Parser einer bestimmten Sprache mit der Codegenerierung einer anderen Sprache zu verbinden, da bei den meisten Projekten der Code historisch bedingt umfangreicher wurde und selten klare Interfaces und APIs definiert wurden. Somit waren auch die Möglichkeiten Code zwischen einzelnen Projekten zu teilen stark begrenzt.

Aus dieser Situation starteten Vikram Adve und Chris Lattner im Jahre 2000 ein Studienprojekt an der University of Illinois. Ziel war es eine Architektur für einen Compiler-Unterbau zu erschaffen, welche eine übergreifende Übersetzung und Optimierung von Code ermöglicht. Das Projekt wurde LLVM genannt, was für Low-Level-Virtual-Machine steht.

Abb 2: Logo von LLVM

„The LLVM Project is a collection of modular and reusable compiler and toolchain technologies“ *– www.llvm.org*

Von Anfang an stand ein Motto für die Entwicklung fest: „llvm is a set of **modern** and **reusable** libraries with well-defined **interfaces**“. Statt auf Eigenentwicklung zu setzen, sollte LLVM nach „State of the Art“-Prinzipien des Compilerbaus aufsetzen. Um Fehler im Aufbau zu vermeiden,

wurde von Beginn darauf geachtet, dass jeder Bestandteil von LLVM gut dokumentiert und nur über APIs genutzt wurde. Ziel war es ein zukunftsfähiges und erweiterbares System zu schaffen.
Dies stellte sich als die richtige Entscheidung heraus, da LLVM heute nicht einfach nur ein Compiler ist, sondern eine komplette Infrastruktur. Sie beinhaltet eine Vielzahl von Unterprojekten und Erweiterungen, von C-Compiler über Debugger und OpenCL-Libaries bis hin zu Java/.NET Virtuals Machines. Aufgrund der o.g. Prinzipien ist es trotzdem weiterhin effektiv Teile von LLVM zu benutzen und für seine Bedürfnisse anzupassen.

Aufbau / Merkmale LLVM

Um die Alleinstellungsmerkmale und die Struktur von LLVM zu erkennen und schätzen zu lernen, ist es hilfreich LLVM mit dem Aufbau eines „klassischen Compilers“ gegenüberzustellen.

„Altbekannter“ Aufbau eines Compilers:

Abb 3: „Altbekannter“ Aufbau eines Compilers

Der bisherige Compileraufbau verlangte von dem Programmieren, das sie Wissen über Frontend und Backend besaßen. Wollte man z.B. ein Frontend für eine neue Sprache für gcc schreiben, musste man, obwohl man mit der späteren Codeerzeugung nur indirekt in Kontakt kam, sich mit speziellen Datenstrukturen aus dem Backend auseinandersetzen. Grund hierfür war, dass das Backend die Daten in spezieller Form erwartete. Das Frontend musste teils auf Funktionen, welche im Backend definiert waren, zugreifen, teils Getter für Daten bereitstellen, da es ein „Hin-und-Her“ zwischen den Aufrufen aus dem Backend ins Frontend bzw. umgekehrt gab.

Unabhängigkeit Frontend – Backend:

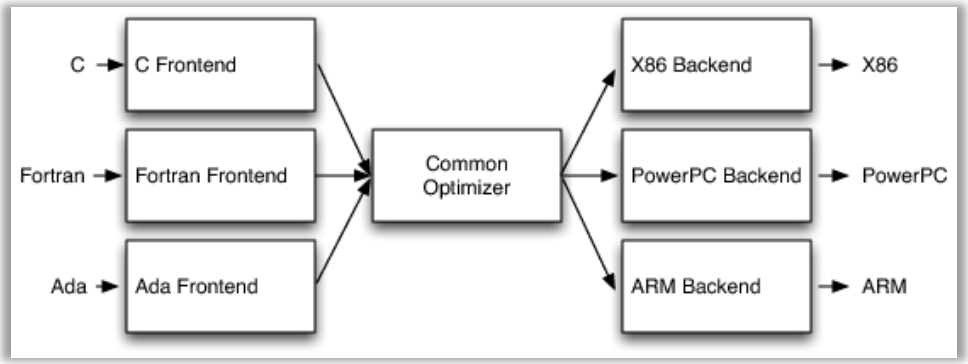

Abb 4: LLVM Aufbau

Auch LLVM ist ein Compiler, welcher in drei Komponenten (Frontend, Common Optimizer, Backend) aufgeteilt ist.

Betrachtet man nun die Unterschiede, lässt sich erkennen, dass LLVM dahingehend optimiert ist, eine gemeinsame Grundlage für möglichst viele Sprachen und Targets zu schaffen. Durch diese Aufteilung ist es nun möglich, „n“ Sprachen allgemein zu Optimieren und dann auf jedes von „m“ beliebige System zu exportieren. Ohne diese Aufteilung wären „n“ * „m“ Compiler notwendig. Durch LLVM existiert nun jedoch nur noch eine Compiler-Infrastruktur, welche man kennen muss. Weiterhin können neu entwickelte Sprachen auf bestehende Optimierungen zurückgreifen, sodass diese von Anfang an performant laufen können.

Jedes LLVM-Frontend, welches den Parser der Sprache enthält und für den Aufbau eines AST zuständig ist, benutzt als Schnittstelle zur Codegenerierung den „Common Optimizer“. Dies dient als Alternative, um nicht für jede Sprache einen eigenen Optimizer entwickeln zu müssen. Wie und wieso dieses Konzept praktikabel ist, wird im nächsten Kapitel beschrieben.

Da es sich in Abb. 4 um einen gerichteten Graphen handelt, soll noch einmal verdeutlicht werden, dass das Frontend in sich abgeschlossen ist. Frontend besitzt keine Kenntnis über Backend. Dies bedeutet jedoch auch, dass der Programmierer eines Frontends diese jedoch auch nicht besitzen muss, sondern nur über eine API mit dem Common Optimizer kommuniziert.

LLVM IR

Damit der Common Optimizer effektiv und effizient arbeiten kann und zudem seine Funktionalität allgemeingültig zur Verfügung stellt, muss eine Repräsentationsform des Programm-Codes gefunden werden, die zur weiteren Verarbeitung geeignet ist. LLVM setzt hierbei auf LLVM IR.

LLVM IR ist eine Kurzform für „Intermediate Repräsentation“ und bezeichnet eine spezielle Repräsentationsform des Programmcodes im Compiler. Die Idee hinter LLVM IR ist eine target-unabhängige Sprache zu erschaffen, welche durch eine wohldefinierte Syntax in der Darstellung beliebig zwischen Text, In-Memory und Binärdarstellung wechseln kann. LLVM IR ist zudem auch sprachunabhängig, was eine sehr wichtige Eigenschaft darstellt, da alle Sprachen die ein LLVM Frontend besitzen, zwangsläufig durch LLVM IR repräsentiert werden. Dieses Verhalten wird dadurch erreicht, dass LLVM IR keine Vorgaben hinsichtlich grundlegender Sprachkonzepte wie z.B. Speichermanagement oder Error-Handling enthält.

LLVM IR ist der Zugang von LLVM zur Code-Optimierung und Code-Generierung. In LLVM IR können Code-Analysen und Optimierungen unabhängig vom Frontend durchgeführt werden. Unter die möglichen Optimierungen fallen z.B.

- Linktime-Optimierungen
- Inteprocedural- Optimierungen (z.B. Erkennung unbenutzte Variablen, toter Code)
- Whole program-Optimierungen (z.B. Erkennung unbenutzte Funktionen)

Da nicht alle Sprachen immer alle Optimierungen benötigen oder gar benutzen können, ist es möglich die anzuwendenden Optimierungen, sogenannte Passes, vom Entwickler auswählen

zu lassen. Folgende Abbildung zeigt den Vergleich von LLVM IR Code vor und nach der Anwendung von beispielhaften Passes.

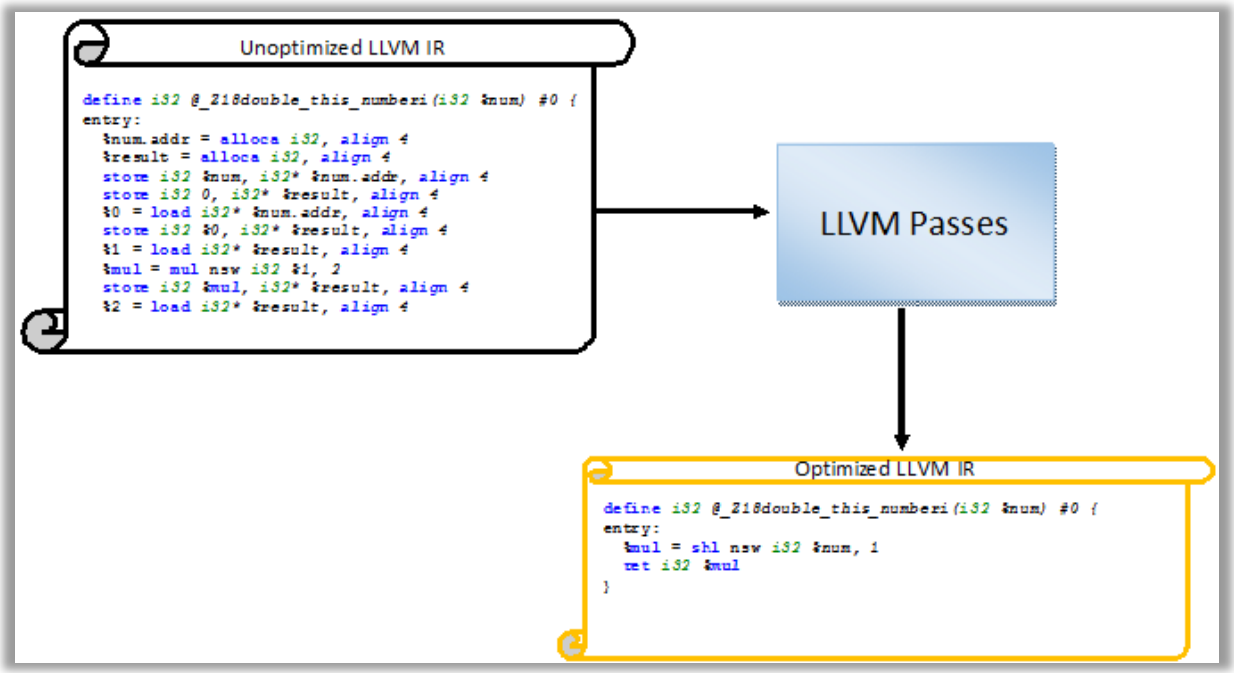

Abb 5: LLVM Passes

Durch die Möglichkeit LLVM IR Code als Text zu speichern und zu einem späteren Zeitpunkt Optimierungen darauf anzuwenden, können ältere Anwendungen für neuere Systeme anders optimiert werden, als sie es eventuell bislang wurden. Beispielsweise speichert Apple von jeder Anwendung im App Store den LLVM IR Code, um bei einem späteren Release eventuell neu zu kompilieren und andere Optimierungen anwenden zu können.

Interpretation eines LLVM IR Beispielcodes

Die Sprache kann als eine Art Assembler-Code angesehen werden und bietet Funktionen, die einen reduzierter Befehlssatz (RSIC) entsprechen. Im Folgenden sollen die wichtigsten Eigenschafften von LLVM IR im Detail besprochen werden.

```
; ModuleID = 'top'
@0 = internal unnamed_addr constant [14 x i8] c"hello world!\0A\00"
define void @main() {
entrypoint:
}
```

Abb 6: LLVM IR

Als Erstes lässt sich festhalten, dass in LLVM IR Register (Variablen) mit @Zahl angesprochen werden. LLVM IR arbeitet mit hierbei mit beliebig vielen Registern, welche automatisch auf die Register, unter Betrachtung der freien Anzahl, des Zielsystems abgebildet werden.

Es ist möglich die virtuellen Register durchzunummerieren, wie im obigen Beispiel (Abbildung 2) oder ihnen einen expliziten Namen zu geben. LLVM IR arbeitet hier nach dem SSA-Prinzip (static single assignment form, siehe Exkurs SSA).

Zusätzlich zu SSA wird eine starke Bindung verwendet. Im Beispielcode (Abbildung 2) wird dem Register ein Wert des Datentyps „14 x i8“ zugewiesen. Die 14 beschreibt die Anzahl der Elemente (Array) und die i8 wie viele Bit verwendet werden (i32 ist beispielsweise ein Integer-Wert).

Die Methoden haben den typischen Aufbau einer Methode, mit dem Zusatz das sie einen „entrypoint:“ enthalten müssen, um LLVM den Einstiegspunkt in der Methode zu verdeutlichen.

In Methoden sind anschließend bekannte Operationen, wie load, store, add oder alloca verfügbar.

Exkurs SSA

SSA beschreibt das Konzept jede Variable nur einmal zu beschreiben (Konstanten, vergleichbar mit „final“ in Java). Durch dieses Konzept wird ein expliziter Datenfluss gewährleistet.

Zur Verdeutlichung ein Beispiel:
y := 1
y := 2;
z := y;

Der Compiler kann in diesem Fall nicht ohne Weiteres feststellen ob redundante Variablen oder Zuweisungen entfernt werden können. Unter Einhaltung des Konzeptes von SSA, erhält jede Variable einen Index.

y1 := 1
y2 := 2;
z1 := y2;

In diesem Fall kann der Compiler problemlos feststellen, das y_1 nie verwendet wird und es somit entfernen.

Dieses Konzept wird durchgängig in LLVM verwendet, von der Auswertung von veränderlichen Variablen, über die Auswertung von if-else-Abfragen, bis hin zu Evaluierung von Schleifen. Um das Konzept zu erleichtern, kann eine SSA-Form fast automatisiert über eine sogenannte „phi instruction” aufgebaut werden.

Projektvorstellung: subset

Abschließend soll auch auf die praktische Anwendung von LLVM eingegangen werden. Um die theoretischen Konzepte zu Verdeutlichen und zur Anwendung zu bringen, wurde ein Frontend für LLVM entwickelt.

Dieses Frontend analysiert eine Sprache, welche sich größtenteils an Element der Sprache C orientiert. Subset bietet hierfür zwei Datentypen (double, int), veränderliche Variablen, arithmetisch bzw. logische Ausdrücke, if-else-Abfragen, Methoden-Deklarationen, sowie den Aufruf von externen Funktionen (printf).
Um jedoch zu verdeutlichen, dass LLVM auch offen für neue Sprachkonzepte ist und diese sich ebenfalls effizient in LLVM IR abbilden lassen, wurde die Grammatik der Sprach so entworfen, dass neue Konzepte Anwendung fanden. Hierzu zählt z.B. das Anweisungen, die in keinem Ausführungskontext, d.h. in keiner Methode stehen, automatisch einen globalen Kontext erhalten und mitausgeführt werden.

Für die lexikalische und syntaktische Analyse wird für dieses Projekt auf altbewehrte Methoden zurückgegriffen. Um den Wurzeln von LLVM zu folgen, wurde flex eingesetzt um einen Scanner zu erzeugen, sowie bison als Parsergenerator.
Die genauen akzeptierten Zeichen und Regeln der Grammatik sind den Dateien tokens.l und parser.y zu entnehmen.

Anschließend nach der syntaktischen Analyse wird ein AST aufgebaut. Dieser enthält wie üblich, alle relevanten Symbole und Informationen um abschließend Code zu erzeugen. Aus Gründen der Vereinfachung wurde auf die Traversierung durch Visitoren verzichtet. Stattdessen enthält jeder Knoten eine LLVM-spezifische Methode: virtual llvm::Value* codeGen(CodeGenContext& context). Durch diese Methode wird in der Klasse codegen der zu produzierendeLLVM IR Code spezifiziert. Durch den Aufbau von Blöcken wird in dieser Klasse zusätzlich ein Control Flow Graph aufgebaut und der Ablauf des Programmes modelliert.

Der entstandene LLVM IR Code wird schlussendlich in die Datei ir.code ausgeben. Der zu kompilierende Porgramm-Code wird in der Datei source.subset spezifiziert.

Durch den Aufruf des LLVM-spezifischen Kommandos opt, kann der erzeugte Code mit spezifischen Optimierungen verssehen werden. Da diese sehr vielfältig sind, wurde auf eine automatisierte Ausführung verzichtet. Durch den Parameter–S kann der optimierte Code wieder als LLVM IR ausgeben werden.

Dieser finale Code kann nun auf allen von LLVM unterstützen Plattformen in Maschinencode umgewandelt und ausgeführt werden.

Beispielhafte Ausführung:

1. make clean → Alte Parser-, Scanner- und Output-Dateien löschen
2. make -> Kompiliere alle Programme + führe llvm aus.

3. Als Input sei folgender Code verwendet:

```
int antwort(int var) {
        double zeitZumBerechnen = 1000000.0;
        printf("%d zeitZumBerechnen\n", zeitZumBerechnen);
        if (var < 42) {
                return -1;
         } else {
                return 42;
        };
};

printf("%d\n", antwort(42));
```

4. Dieser Code wird durch folgenden LLVM IR Code repräsentiert:

```
; ModuleID = 'main'
@.str0 = private constant [22 x i8] c"%d zeitZumBerechnen\0B\0D\00"
@.str1 = private constant [5 x i8] c"%d\0B\0D\00"
define void @main() {
entry:
  %0 = call i32 @antwort(i32 42)
  %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32 0,
i32 0), i32 %0)
  ret void
}
declare i32 @printf(i8*, ...)
declare void @putchar()
define i32 @antwort(i32 %var1) {
entry:
  %var = alloca i32
  store i32 %var1, i32* %var
  %zeitZumBerechnen = alloca double
  store double 1.000000e+06, double* %zeitZumBerechnen
  %0 = load double* %zeitZumBerechnen
  %1 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([22 x i8]* @.str0, i32 0,
i32 0), double %0)
  %2 = load i32* %var
  %3 = icmp slt i32 %2, 42
  br i1 %3, label %branch0, label %branch1

branch0:                                          ; preds = %entry
  %4 = sub i32 0, 1
  ret i32 %4
branch1:                                          ; preds = %entry
  ret i32 42
}
```

5. Durch den Aufruf von opt -S -Oz ir.code kann der Code optimiert werden:

```
; ModuleID = 'main'

@.str0 = private constant [22 x i8] c"%d zeitZumBerechnen\0B\0D\00"
@.str1 = private constant [5 x i8] c"%d\0B\0D\00"

; Function Attrs: nounwind
define void @main() #0 {
entry:
  %0 = tail call i32 @antwort(i32 42)
  %1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([5 x i8]* @.str1, i32
0, i32 0), i32 %0)
  ret void
}

; Function Attrs: nounwind
declare i32 @printf(i8* nocapture readonly, ...) #0

; Function Attrs: nounwind
define i32 @antwort(i32 %var1) #0 {
entry:
  %0 = tail call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([22 x i8]* @.str0,
i32 0, i32 0), double 1.000000e+06)
  %1 = icmp slt i32 %var1, 42
  %. = select i1 %1, i32 -1, i32 42
  ret i32 %.
}

attributes #0 = { nounwind }
```