LLVM als IR

BC George (HSBI)

Motivation

Motivation

Es ist ein neuer Prozessor entwickelt worden mit einem neuen Befehlssatz, und es sollen für zwei Programmiersprachen Compiler entwickelt werden, die diesen Befehlssatz als Ziel haben. Was tun?

Themen für heute: Letzte Phase in Compiler-Pipeline

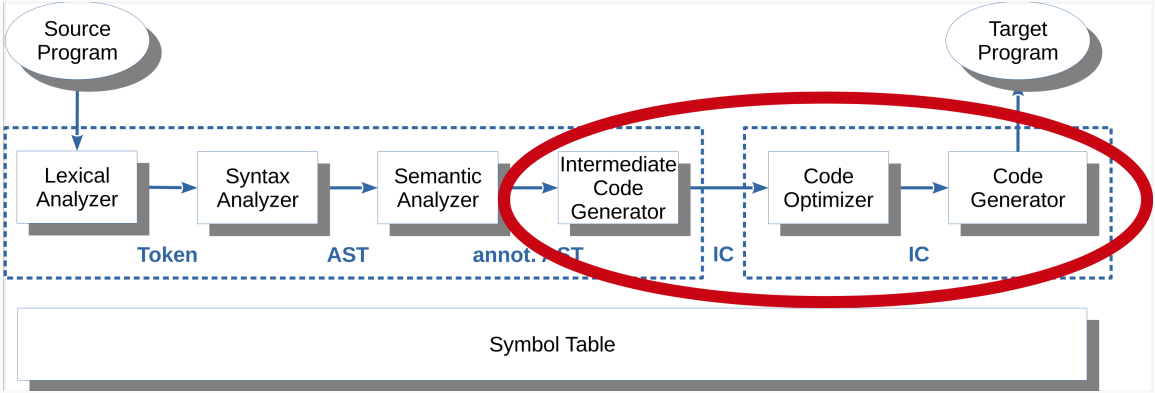


Abbildung 1: Compiler-Pipeline

LLVM - Ein Überblick

Was ist das Ziel von LLVM?

LLVM ist ein Open-Source-Framework, welches die modulare Entwicklung von Compilern und ähnlichen sprachorientierten Programmen ermöglicht. Kernstücke sind LLVM IR (eine streng typisierte Zwischensprache), ein flexibel konfigurierbarer Optimierer, der zur Compilezeit, Linkzeit und Laufzeit eingesetzt werden kann und ein Codegenerator für zahlreiche Architekturen.

Mit LLVM lassen sich sowohl AOT- als auch JIT-Compiler entwickeln. Die Idee ist, (mit Generatoren) ein Frontend zu entwickeln, das Programme über einen AST in LLVM IR übersetzt, und dann mit Hilfe der LLVM-Bibliotheken Maschinencode oder VM-Code zu generieren. Die Komponenten des Compilers sind entkoppelt und kommunizieren über APIs (Unabhängigkeit).

Der Vorteil: Um n Sprachen für m Architekturen zu übersetzen, muss man bestenfalls n Frontends und m Codegeneratoren entwickeln, statt $n \times m$ Compiler zu schreiben.

Der Werdegang von LLVM

- ab 2000 Forschungsprojekt zur Untersuchung dynamischer Kompilierung und Optimierungen von Prof. Vikram Adve an der University of Illinois
- 2002 Masterarbeit von Chris Lattner; "LLVM: An Infrastructure for Multi-Stage Optimization"
  - Siehe auch: LattnerMSThesis
- Kern des Projekts: LLVM IR und Infrastruktur
- ursprünglich **L**ow **L**evel **V**irtual **M**achine, wird aber nicht mehr als Akronym gesehen
- Chris Lattner ist weiterhin der führende Kopf des LLVM-Projekts

Was ist daraus geworden?

Open Source - Framework (in C++ geschrieben) für die Entwicklung von

- Debuggern
- JIT-Systemen
- AOT-Compilern
- virtuellen Maschinen
- Optimierern
- Systemen zur statischen Analyse
- etc.

Für das LLVM-Projekt haben 2012 Vikram Adve, Chris Lattner und Evan Chang den ACM Software System Award bekommen.

Wer setzt es ein?

Adobe AMD Apple ARM Google

IBM Intel Mozilla Nvidia Qualcomm

Samsung ...

Komponenten (Projekte) von LLVM

Der LLVM-Kern

LLVM Core: Optimierer und Codegenerator für viele CPU- und auch GPU-Architekturen

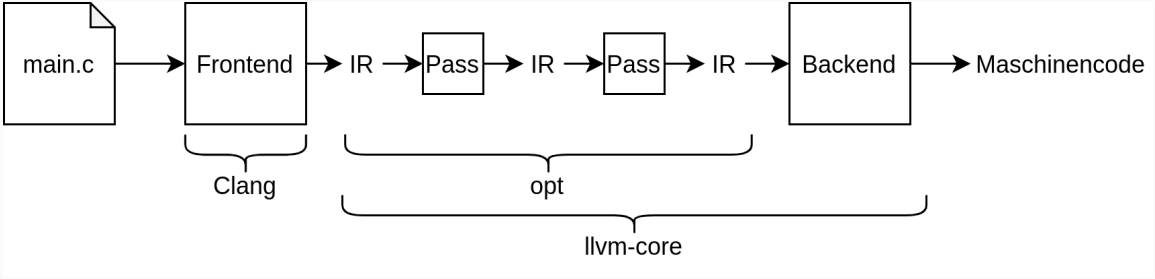
- Optimierer arbeitet unabhängig von der Zielarchitektur (nur auf der LLVM IR)
- sehr gut dokumentiert
- verwendete Optimierungspässe fein konfigurierbar
- Optimierer auch einzeln als Tool `opt` aufrufbar
- wird für eigene Sprachen als Optimierer und Codegenerator eingesetzt

Der Compiler Clang

Clang: C/C++/Objective-C - Compiler auf Basis von LLVM mit aussagekräftigen Fehlermeldungen und Warnungen

- Plattform zur Entwicklung von Source Level Tools
- **Clang Static Analyzer**: Teil von Clang, aber auch separat zu benutzen
- **Clang tidy**: analysiert ebenfalls Code statisch, einzelne Checks konfigurierbar

Clang Toolchain



Die Compiler-Runtime-Bibliothek

compiler-rt: generiert Code, der zur Laufzeit benötigt wird:

- **built-ins**: eine Bibliothek optimierter Implementierungen von Low-Level-Funktionen
  - z.B. die Konvertierung von double nach int 64 auf einer 32-Bit-Architektur
- **profile**: eine Bibliothek, die Informationen über Reichweiten (coverage information) erfasst
- **BlocksRuntime**: implementiert maschinenunabhängig die Runtime-Schnittstellen von Apple "Blocks" (Objective-C)
- **Sanitizer Runtimes**: Laufzeitbibliotheken, welche für die Einbettung und Verwendung von Sanitizern benötigt werden

Die Sanitizer in compiler-rt

Sanitizer sind eine Methode zur Instrumentierung (Code der in das kompilierte Programm eingebettet wird), um die Lokalisierung und Analyse von verschiedensten Fehlerquellen zu erleichtern.

Mithilfe der Sanitizer in der compiler-rt können z. B. Speicherfehler, Race Conditions, Speicherlecks und viele weitere Fehlerquellen gefunden und analysiert werden.

Weitere Komponenten von LLVM

LLDB: Debugger innerhalb des LLVM-Systems

libc++ und libc++ABI: hochperformante Implementierungen der C++-Standardbibliothek, auch für C++11 und C++14

OpenMP: eine zu linkende OpenMP-Bibliothek für Shared-Memory-Programmierung (Multiprozessorcomputer)

polly: spezielle Schleifenoptimierungen auf Polyeder-Basis

libclc: eine Bibliothek für OpenCl, um nichtgrafische Anwendungen auf Grafikprozessoren zu nutzen

klee: zur automatischen Testgenerierung

LLD: ein Linker

und viele weitere Tools, z. B. zum Testen von Compilern.

Externe LLVM-Projekte

Für folgende Sprachen gibt es Compiler oder Anbindungen an LLVM (neben Clang):

Cuda Go Haskell Java Julia Kotlin

Lua Numba Python Ruby Rust Swift ...

Für weiter Projekte siehe Projects built with LLVM

LLVM IR

LLVM IR

- menschenlesbarer Zwischencode
- generische Maschinensprache
- Speicherung von Variablen stackbasiert oder in Registern
- Register können entweder nummeriert werden oder explizite Namen bekommen
- Register-Referenz: `%1`, Variablen-Referenz: `@1`

LLVM IR (cnt.)

- Instruktionsumfang der IR an RISC-Befehlssatz angelehnt (**R**educed **I**nstruction **S**et **C**omputer)
- IR ist immer in **S**ingle **S**tatic **A**ssignment-Form
- streng typisiert
- keine Vorgaben bzgl. grundlegender Sprachkonzepte, wie z.B. Speichermanagement, Error Handling
- Durchgängige Verwendung von Kontrollflussgraphen zur Unterstützung des Optimierers

Vgl. auch: LLVM Dev Conference

Darstellungsformen von LLVM IR

LLVM IR existiert in drei Formen, die äquivalent sind und ineinander überführt werden können:

- menschenlesbar als Text (`.ll`)
- Bitcode (`.bc`)
- In-Memory Representation zur Programmlaufzeit des Compilers

(Die clang-Option -S gibt die menschenlesbaren LLVM IR aus.)

Typen in LLVM IR

LLVM IR-Typen sind plattformunabhängig und lassen sich direkt in optimalen Maschinencode übersetzen.

```
i1 1  ;                    boolean bit
i32 299792458 ;            integer
float 7.29735257e-3 ;      single precision
double 6.62606957e-34 ;    double precision
[10 x float] ;             Array of 10 floats
[10 x [20 x i32]] ;        Array of 10 arrays of 20 integers
<8 x double> ;             Vector of 8 double
```

SSA-Form (Static single assignment)

- bestimmte Form der Intermediate Representation
- jede Variable muss definiert sein, bevor sie verwendet wird
- jeder Variablen wird nur **einmal** ein Wert zugewiesen
- anschließend wird nur noch lesend auf die Variable zugegriffen
- erleichtert die Realisierung vieler Optimierungsverfahren

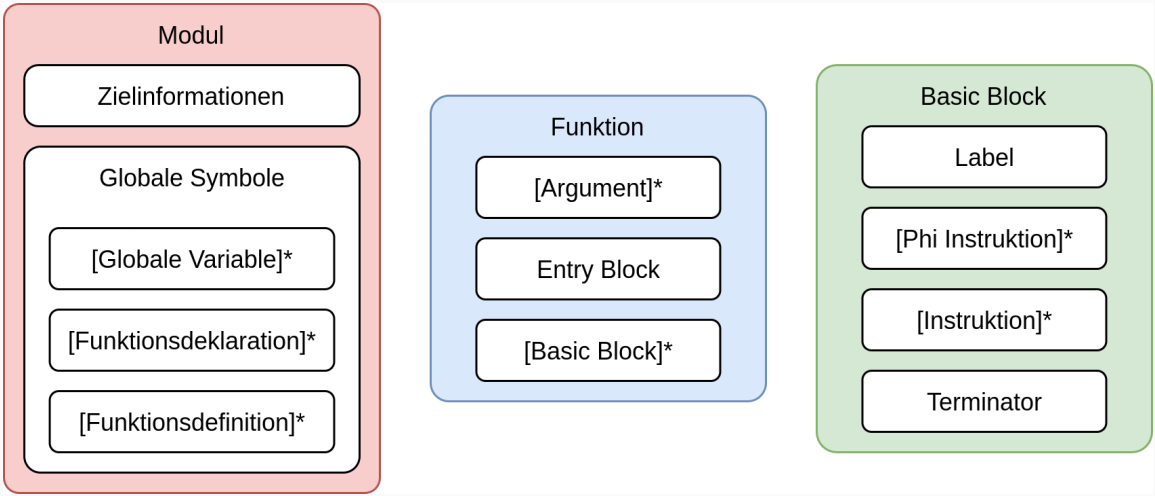
SSA-Form (Beispiel)

Ein Beispiel:

```
// nicht SSA          // SSA
y := 1                y1 := 1
y := 2          =>    y2 := 2
x := y                x1 := y2
```

Hierarchie der LLVM IR

Die LLVM IR ist hierarchisch aufgebaut:



Instruktionen

Operationen:

- arithmetische Operationen
- Vergleichsoperationen
- Kontrollflussoperationen
- Block-Terminatoren

Instruktionen haben keinen oder einen Rückgabewert

- Das Ergebnis hat ein eindeutiges Label (SSA)

Beispiel für eine `add`-Instruktion:

```
%3 = add nsw i32 %3, 10
```

Basic Blocks

Aufbau von Basic Blocks:

- Label (erforderlich)
- Phi Instruktionen
- Instruktionen
- Terminator (erforderlich)
  - (bedingte) Sprunginstruktion
  - Returninstruktion

Basic Blocks (Beispiel)

Beispiel für einen Basic Block:

```
2:                                ; label
  %3 = add nsw i32 2, 20          ; instruktion
  %4 = icmp sgt i32 %3, 10        ; instruktion
  br i1 %4, label %5, label %7    ; terminator
```

Funktionen

Aufbau von Funktionen:

- Argumente
- Entry Block (erforderlich): erster Basic Block, der immer ausgeführt wird
- weitere Basic Blocks

Funktionen: Beispiel

```c
// func.c

long f(long a, long b){
  long x = 1;
    if (a > b)
      x += 20;
    else
      x += b;
    return x;
}
```

Ausgabe von entsprechender LLVM IR:

```
clang -O3 -opt -Xclang -disable-llvm-passes
    -S -emit-llvm func.c -o func.ll

opt -S -mem2reg -instnamer func.ll -o func_mem2reg.ll
```

Funktionen: Beispiel in IR

```
; func_mem2reg.ll
define dso_local i64 @f(i64 %arg, i64 %arg1) #0 {
bb:                                               ; entry-block
  %i = icmp sgt i64 %arg, %arg1
  br i1 %i, label %bb2, label %bb4

bb2:                                              ; preds = %bb
  %i3 = add nsw i64 1, 20
  br label %bb6

bb4:                                              ; preds = %bb
  %i5 = add nsw i64 1, %arg1
  br label %bb6

bb6:                                              ; preds = %bb4, %bb2
  %.0 = phi i64 [ %i3, %bb2 ], [ %i5, %bb4 ]
  ret i64 %.0
}
```

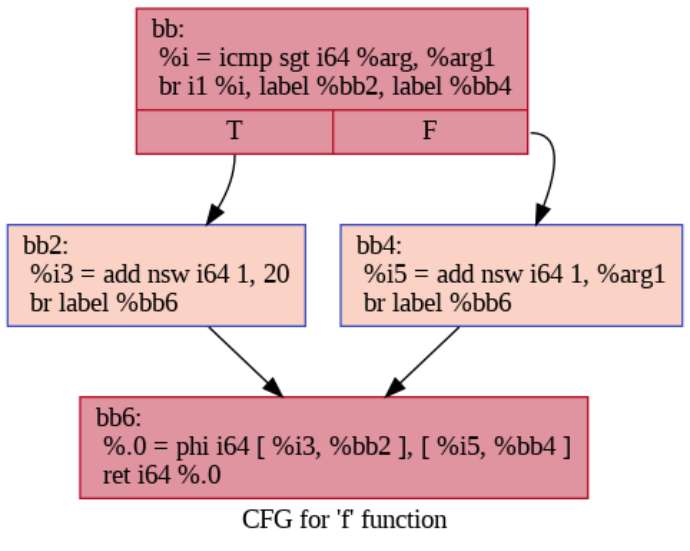
Kontrollflussgraph

- Die Basic Blocks einer Funktion sind durch Sprunginstruktionen verbunden
- Basic Blocks haben Vorgänger und Nachfolger
- Der so entstehende Kontrollfluss wird von LLVM ständig durch einen Kontrollflussgraphen modelliert

Ausgabe des Kontrollflussgraphen im `.dot` Format:

```
opt -dot-cfg func_mem2reg.ll > /dev/null
```

Kontrollflussgraph (Ergebnis)



Essentiell: die Phi-Instruktion

- Basierend auf dem Programmfluss können einer Variablen unterschiedliche Werte zugewiesen werden (`x` im vorherigen Beispiel)
- SSA: jeder Variablen / jedem Register wird nur einmal ein Wert zugewiesen
- Daher: virtuelle Register entsprechen nicht 1:1 den Variablen des kompilierten Progamms
- Der Wert einer konditionell zugewiesenen Variablen kann daher in zwei unterschiedlichen virtuellen Registern stehen (je nach durchlaufenem Programmzweig)
- Problem: welches Register soll gelesen werden, um den Wert der Variablen weiter zu verwenden?
- Lösung: Phi-Instruktion speichert Werte in einem neuen Register abhängig vom Vorgängerblock

Phi-Instruktion: Beispiel

```
bb6:                                              ; preds = %bb4, %bb2
  %.0 = phi i64 [ %i3, %bb2 ], [ %i5, %bb4 ]
  ret i64 %.0
```

Effekt: Speichert den Wert aus Register `%i3` in Register `%.0`, falls wir aus Block `%bb2` kommen, sonst wird der Wert aus Register `%i5` gespeichert

Module

Bildet eine Übersetzungseinheit eines Programms ab

Inhalt:

- Ziel Informationen (erforderlich)
  - Datenlayout (Endianess, native Integergrößen, etc.)
  - Ziel-Triplet (Zielarchitektur, ABI, etc.)
- Globale Variable
- Funktionsdeklarationen
- Funktionsdefinitionen

Module werden vom Linker zum lauffähigen Programm zusammengefügt

Ein weiteres Beispiel

```c
int main() {
    int x = 7;
    int y = x + 35;

    return 0;
}
```

```
clang -emit-llvm -S hello.c
```

So sieht LLVM IR dafür aus

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 7, i32* %2, align 4
%4 = load i32, i32* %2, align 4
%5 = add nsw i32 %4, 35
store i32 %5, i32* %3, align 4
ret i32 0
```

Entsprechender Assembler-Code Teil 1

(Ausgabe ohne `-emit-llvm -S` Optionen)

```
    .text
    .file    "hello.c"
    .globl    main                    # -- Begin function main
    .p2align    4, 0x90
    .type    main,@function
main:                                   # @main
    .cfi_startproc
# %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
```

Entsprechender Assembler-Code Teil 2

```
    xorl    %eax, %eax
    movl    $0, -4(%rbp)
    movl    $7, -8(%rbp)
    movl    -8(%rbp), %ecx
    addl    $35, %ecx
    movl    %ecx, -12(%rbp)
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
.Lfunc_end0:
    .size    main, .Lfunc_end0-main
    .cfi_endproc
                                        # -- End function

    .ident    "clang version 9.0.1 "
    .section    ".note.GNU-stack","",@progbits
    .addrsig
```

Der LLVM-Optimierer

Allgegenwärtig in LLVM: Der Optimierer

- Teil von LLVM Core
- kann zur Laufzeit, Compilezeit und Linkzeit eingesetzt werden
- nutzt auch Leerlaufzeit des Prozessors
- läuft in einzelnen Pässen über den Code. Jeder Pass kann einzeln aufgerufen und konfiguriert werden.
- generiert in der Regel sehr schnellen Code
- arbeitet auf Basic Blocks und DAGs

Arten von Pässen

Analysis passes sammeln Informationen für andere Pässe, z. B. zum Debuggen oder Visualisieren

Transform passes verändern das Programm auf irgendeine Art und Weise

Utility passes sind Hilfspässe, z. B. die Umformung des IR in Bitcode

Vgl. auch: LLVM's Analysis and Transform Passes

Einige Optimierungen in LLVM

- Dead Code Elimination
- Aggressive Dead Code Elimination
- Dead Argument Elimination
- Dead Type Elimination
- Dead Instruction Elimination
- Dead Store Elimination
- Dead Global Elimination

Vgl. auch: LLVM's Analysis and Transform Passes

SSA und Optimierung

Der Optimierer profitiert stark von SSA. Der Wert einer Variablen wird nicht überschrieben, sodass sie niemals in der *kill*-Menge der Datenflussanalyse vorkommt, wodurch die Datenflussanalyse in vielen Punkten erleichtert wird.

Die Codegenerierung

Struktur der Codegenerierung

- übersetzt LLVM IR in Maschinencode
- läuft in Pässen:
  - Built-In-Pässe, die defaultmäßig laufen
  - Instruction Selection
  - Register Allocation
  - Scheduling
  - Code Layout Optimization
  - Assembly Emission

Unterstützte Prozessorarchitekturen

x86 AMD64 PowerPC PowerPC 64Bit Thumb

SPARC Alpha CellSPU PIC16 MIPS

MSP430 System z XMOS Xcore ...

Wrap-Up

Wrap-Up

- LLVM ist eine (fast) komplette Infrastruktur zur Entwicklung Von Compilern und compilerähnlichen Programmen. Die wichtigsten Bestandteile sind der Zwischencode LLVM IR und der LLVM Optimierer.

LICENSE