LLVM IR

Enea Zaffanella

enea.zaffanella@unipr.it

16 novembre 2020

Linguaggi, interpreti e compilatori
Laurea Magistrale in Scienze informatiche

Struttura di un file di bitcode LLVM

- Un file di bitcode LLVM rappresenta un **modulo**
  - (ottenuto dalla "compilazione" di una unità di traduzione)
- Un modulo può contenere:
  - dichiarazioni o definizioni di **variabili** globali
  - dichiarazioni o definizioni di **funzioni** globali
  - **metadata** (per ottimizzazione o debugging)
- Ogni dichiarazione o definizione ha un **linkage**
  - `external`, `weak`, `internal`, ...

Visualizzazione bitcode LLVM in formato testo

Partendo dal sorgente:

- `clang [options] -S aaa.c -emit-llvm -o aaa.ll`
- (`llvm-as aaa.ll -o aaa.bc`)
- considerare opzione clang `-fno-discard-value-names`

Partendo dal bitcode LLVM precompilato:

- (`clang [options] -c aaa.c -emit-llvm -o aaa.bc`)
- `llvm-dis aaa.bc -o aaa.ll`

Identificatori

- per riferirsi a **tipi** e **valori** (variabili, funzioni, etichette, . . .)
- **globali**: il nome inizia con il carattere `@`
- **locali**: il nome inizia con il carattere `%`
- identificatori "anonimi": il nome è un intero non negativo
  - `@123`, `%0`, `%1`, `%2`
- identificatori con nome usano la RE
  `{[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*}`
  - `@x`, `@main`, `@bond.james.bond.007`

Tipi di dato semplici

- void (nessun valore): `void`
- interi (con dimensione N in bits): `i`N
  - `i1`, `i8`, `i23`, `i32`, `i123456`
- floating point (dimensioni standard)
  - `half`, `float`, `double`, `fp128`
  - anche specifici: `x86_fp80`, `ppc_fp128`
- puntatori: *type* `*`
  - `i32*`
- `label`, `token`

Tipi di dato composti

- array: [*n* x *type*]
  - `[10 x i32]`
  - `[10 x [20 x i8]]`
- vector: < *n* x *type* >
- struct: { *typelist* }
  - `{ i32, i8*, float, [5 x i32] }`
  - `<{ i16, i8, i32 }> ; packed`
- funzioni: *ret-type* (*typelist*)
  - `i32 (int32, i1)`
  - `i32 (int8*, ...)`
- `metadata`, ...

Target triple e data layout

- `target triple = "x86_64-pc-linux-gnu"`
  `target datalayout = "`*`layout spec`*`"`
- esempio layout:
  `"e-m:e-i64:64-f80:128-n8:16:32:64-S128"`
- `e` o `E` ⇒ little o big endianness
- requisiti e preferenze di alignment per i valori dei vari tipi:
  - `i`*`N`*`:`*`abi`*`:`*`pref`* ⇒ interi
  - `f`*`N`*`:`*`abi`*`:`*`pref`* ⇒ floating point
  - `v`*`N`*`:`*`abi`*`:`*`pref`* ⇒ vector
  - `p:`*`abi`*`:`*`pref`*`:`*`idx`* ⇒ puntatori (*idx* è la size degli indici)
  - `a:`*`abi`*`:`*`pref`* ⇒ aggregati (array e struct)
  - `S:`*`size`* ⇒ stack
- `n8:16:32:64` ⇒ size degli interi "nativi" del target
- `m:e` ⇒ mangling dei nomi (ELF style)

Costanti

- booleane (tipo `i1`): `true`, `false`
- intere: 4, 56, -1234 (anche negative)
- floating point: `123.0`, `1.5e12`
  (solo esatte: `1.3`, `0.1` non sono valide)
- puntatore: `null`, `@global`
- array: `[ i32 3023, i32 -12, i32 18 ]`
- vector: `< i32 3023, i32 -12, i32 18, i32 4096 >`
- struct: { `i32 3, float 2.5, i32* @global` }
- `zeroinitializer`
- `undef`

Definizioni

Definizioni di tipo

```
%point_t = type { int32, int32 }
%my_array_t = type [100 x %point_t]
```

Definizione di costante/variabile globale

```
@ottavo = internal constant double 0.125, align 8
@counter1 = dso_local global i32 0, align 4
@array = dso_local global %my_array_t zeroinitializer, align 16

@.str = private constant [6 x i8] c"Hello\00", align 1
```

Definizioni di funzioni

Dichiarazione pura

```
declare dso_local void @bar()
```

Definizione

```
define dso_local i32 @main() #0 {
  call void @bar()
  ret i32 0
}
```

La struttura di una funzione

- una funzione è una sequenza di **basic block** (BB)
- un BB è una sequenza di **istruzioni**
- solo l'ultima istruzione del BB (terminatore) cambia il flusso di esecuzione (determina il BB successivo o l'uscita dalla funzione)
- il codice è in forma SSA (Static Single Assignment form)
- per generare i nomi locali al BB (tipi, valori, etichette) si usa un unico contatore (poco leggibile)
- l'opzione `-fno-discard-value-names` cerca di mantenere i nomi locali (più leggibile)

Istruzioni LLVM

- ne consideriamo solo un sottoinsieme
  - no gestione eccezioni
  - no gestione parallelismo
  - no funzioni intrinseche
- classificazione
  - istruzioni aritmetiche e bit-a-bit
  - istruzioni che operano/indirizzano la memoria
  - istruzioni di confronto
  - terminatori dei basic block
  - conversioni di tipo
  - chiamate di funzione

Istruzioni aritmetiche

Operatori binari

- `add`, `sub`, `mul`, `sdiv`, `udiv`, `srem`, `urem`
- `fadd`, `fsub`, `fmul`, `fdiv`, `frem`
- `%2 = add i32 %0, %1`
  `%3 = add i32 %2, 15`
- modificatori `nsw`, `nuw`

Operatori unari

- `fneg` (negazione per tipi floating point)
  `%2 = fneg float %1`
- Negazione sui tipi interi?
  `%1 = sub nsw i32 0, %0`

Istruzioni bit-a-bit

Operatori binari

- `and`, `or`, `xor`
- `%2 = or i1 %0, %1`
  `%4 = and i32 %3, 255`
- shift: `shl`, `lshr`, `ashr`
- `%0 = shl i8 31, 4`

Negazione logica e bit-a-bit?

- `%1 = xor i1 %0, true`
- `%3 = xor i32 %2, -1`

Istruzioni di confronto

Confronti tra interi

- `%res = icmp eq i32 %a, %b`
- producono un valore di tipo `i1` (booleano)
- 10 tipologie di confronto:
  - eq, ne
  - slt, sle, sge, sgt
  - ult, ule, uge, ugt

Confronti tra floating point

- `%res = fcmp eq float %a, %b`
- 16 tipologie di confronto:
  - oeq, one, olt, ole, oge, ogt, ord
  - ueq, une, ult, ule, uge, ugt, uno
  - true, false

Istruzioni di confronto

Selezione (aka conditional move)

- `%res = select i1 %cond, i32 %val0, i32 %val1`
- copia in `%res` il valore `%val0` (risp., `%val1`) se la condizione `%cond` è true (risp., false)

Istruzioni che terminano i blocchi

Return (con o senza valore)

`ret` *type value*

`ret void`

Branch (condizionati o meno)

`br i1` *cond*, `label` *lab1*, `label` *lab2*

`br label` *lab* *; unconditional*

Switch (su valori di tipo intero)

`switch` *type val*, `label` *defaultdst* [*type val*, `label` *dst* ...]

Istruzioni per memoria e indirizzamento

Allocazione

- `alloca`: allocazione sullo stack (scalari o array)
- `%ptr = alloca i32`
  `%ptr = alloca i8, i32 100`
- non inizializzata (`undef`), deallocazione automatica

Load e store

- `%res = load i32, i32* %ptr`
- `store i32 42, i32* %ptr`

Indirizzamento in array e struct

- `%eptr = getelementptr [4 x i32], [4 x i32]* @aptr, i64 2`
- `%fptr = getelementptr {i32, i8}, {i32, i8}* @sptr, i32 1`

Conversioni di tipo (1)

Troncamenti e estensioni

- `%res = trunc i32 %arg to i8`
- `%res = fptrunc double %arg to float`
- `%res = zext i16 %arg to i32`
- `%res = sext i16 %arg to i32`
- `%res = fpext float %arg to double`

Conversioni intero-floating

- `%res = fptosi float %arg to i32`
- `%res = fptoui float %arg to i32`
- `%res = sitofp i32 %arg to float`
- `%res = uitofp i32 %arg to float`

Conversioni di tipo (2)

Puntatore-intero

- `%val = ptrtoint i32* %ptr to i64`
- `%ptr = inttoptr i64 %val to i32*`

Bitcast

- `%res = bitcast i32 %val to [4 x i8]`
- `%res = bitcast %Dtype* %ptr to %Btype*`

Chiamata di funzione

Chiamata di funzione (nothrow)

- `%val = call i32 @sum(i32 %arg1, i32 %arg2)`
- si possono specificare calling convention, alcune ottimizzazioni applicabili e altri attributi
- Nota Bene: per chiamare una funzione che può generare una eccezione occorre usare la `invoke` (terminatore per il BB)

Le funzioni ϕ

- `%val = phi i32 [%val0, %bb0], [%val1, %bb1]`
- sono *pseudo* funzioni, inserite dalla forma SSA
- sono poste all'inizio dei blocchi (prima delle altre istruzioni)
- selezionano un valore tra n alternative ($n = 2$ nell'esempio)
- n è il numero di predecessori del blocco
- le m funzioni ϕ in ingresso al blocco sono eseguite in **parallelo**

Visualizzazione del CFG di una funzione

Creazione del file bitcode per la funzione

```
clang [...] -fno-discard-value-names -S unit.c -emit-llvm -o unit.ll
llvm-extract -S -func='foo' -o foo.ll unit.ll
```

Nota: attenzione al mangling del nome di funzione

Visualizzazione della funzione

- occorre installare il package graphviz
- `opt -view-cfg foo.ll -o /dev/null`
- alternativa: creazione file foo.dot
  `opt -dot-cfg foo.ll -o /dev/null`
- generazione file grafico (vari formati):
  `dot -Tpdf foo.dot -o foo.pdf`