

Linguaggi e compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2023/2024

1 Complementi

- Sul codice C++ usato per generare Human-Readable Intermediate Representation LLVM
 - Metodi di IRBuilder
 - Blocchi, moduli ed emissione del codice IR “human readable”
- Grammatica base per il progetto di primo livello

Linguaggi e compilatori

1 Complementi

- Sul codice C++ usato per generare Human-Readable Intermediate Representation LLVM
 - Metodi di IRBuilder
 - Blocchi, moduli ed emissione del codice IR “human readable”
- Grammatica base per il progetto di primo livello

Istruzioni API C++

- In questo set finale di slide presentiamo un (ristretto) sottoinsieme delle funzionalità disponibili, attraverso le API LLVM, in linguaggio C++, per la generazione di codice in LLVM IR.
- Descriveremo comunque tutti i metodi utilizzati nel corso per la scrittura del front-end per la versione di Kaleidoscope sviluppata.
- Le funzionalità di LLVM sono presentate, organizzandole secondo
 - Generazione delle istruzioni (metodi di IRBuilder)
 - Gestione dei blocchi e del “punto di inserimento” (metodi della classe Function, metodi statici di BasicBlock e ancora metodi di IRBuilder)
 - Operazioni a livello di *modulo* (funzioni e variabili globali)

Metodi di IRBuilder

- La maggior parte delle istruzioni generate mediante IRBuilder lascia il risultato in un registro SSA.
- Il programmatore che utilizza le API LLVM deve sapere che un'istruzione che “scrive” in un registro SSA è identificabile con il registro stesso e questo è rappresentato mediante (un riferimento) ad un oggetto della classe Value
- L'uso “tipico” è quindi:

```
Value *V = <builder>->Create...
```

anche se, nel seguito, il left-value dell'assegnamento non verrà mostrato
- Fanno chiaramente eccezione istruzioni, come quelle di salto, che non producono un risultato da memorizzare

Metodi di IRBuilder

- Le due istruzioni “base” sono chiaramente load e store
 - `<builder>->CreateLoad(<type>,<LocalV>,<Name>)`
 - `<builder>-> CreateLoad(<type>,<GlobalV>,<Name>)`
 - `<builder>->CreateStore(<value>, <Variable>)`

Nel caso della load, il tipo (primo parametro) può essere recuperato usando metodi diversi a seconda che la variabile sia locale o globale:

- `Type *<type> = <LocalV>->getAllocatedType()`
- `Type *<type> = <GlobalV>->getValueType()`

Nel caso della store, `<Variable>` può essere locale o globale. Il parametro `<value>` è di tipo `Value*`,

- Le 4 operazioni binarie che abbiamo usato hanno tutte la stessa struttura:
 - `<builder>->CreateFAdd(<L>,<R>,<Label>)`
 - `<builder>->CreateFSub(<L>,<R>,<Label>)`
 - `<builder>->CreateFMul(<L>,<R>,<Label>)`
 - `<builder>->CreateFDiv(<L>,<R>,<Label>)`

dove `<L>` ed `<R>` indicano registri SSA.

Istruzioni API C++

- Le due sole istruzioni di salto condizionato che abbiamo usato sono indicate di seguito

- `<builder>->CreateFCmpULT(<L>,<R>,<Label>)`
- `<builder>->CreateFCmpUEQ(<L>,<R>,<Label>)`

dove le lettere LT e EQ indicano rispettivamente le relazioni minore (Less Than) e uguale (Equal), riferite al valore nei due registri L ed R. Per gli altri confronti le lettere sono: GT (maggiore), LE (minore o uguale), GE (maggiore o uguale) e NE (non uguale). Il risultato del confronto, un valore *booleano*, viene memorizzato nel registro SSA identificato da `<Label>` (seguito da un numero progressivo).

- Il confronto è normalmente seguito da un'istruzione per “testarne” il risultato (il valore booleano indicato in precedenza):

- `<builder>->CreateCondBr(<Cond>, <TrueBB>, <FalseBB>)`

dove `<Cond>` è il registro con il risultato del confronto mentre `<TrueBB>` e `<FalseBB>` denotano *blocchi di base* dove procede il flusso di controllo nel caso il valore di `<Cond>` sia vero oppure falso.

Metodi di IRBuilder

- La generazione dell'istruzione di salto incondizionato richiede invece, chiaramente, solo la specifica del *Basic Block* di destinazione:
 - `<builder>->CreateBr(<DestBB>)`
- La particolare istruzione PHI consente di assegnare un valore ad un registro in un punto del programma dove è necessario “riunire” più possibili flussi di esecuzione. Si tratta di un meccanismo dell'IR LLVM che agevola enormemente la traduzione dei tipici costrutti di programmazione strutturata (condizionali e iterazioni). La scrittura
 - `<PHINode> = <builder>->CreatePHI(<type>, <N>, <Label>)`dove `<type>` è il tipo (comune a tutte le `<N>` alternative), `<N>` è il numero di possibili flussi entranti e `<Label>` indica, al solito, il registro SSA in cui verrà memorizzato il valore selezionato, genera un *nodo PHI* “vuoto”, al quale andranno aggiunte le `<N>` coppie, ciascuna composta dal valore da selezionare (un registro SSA) e dalla corrispondente sorgente del flusso di esecuzione (un Basic Block):
 - `<PHINode>->addIncoming(<value>, <block>)`

Metodi di IRBuilder

- La gestione delle chiamate di/ritorno da funzioni è (come accennato a lezione) un meccanismo di astrazione piuttosto elevata in LLVM IR, rispetto ad un linguaggio assembler. A livello di API C++, per la chiamata è disponibile l'istruzione

- `<builder>->CreateCall(<CalleeF>, <ArgsV>, <Label>)`

in cui `<CalleeF>` è il nome della funzione, `<ArgsV>` il vettore degli argomenti (registri SSA che memorizzano il valore degli argomenti) e `<Label>` il nome che verrà attribuito al registro SSA con il valore restituito. Per generare l'istruzione di ritorno è invece disponibile la funzione

- `<builder>->CreateRet(<RetVal>)`

in cui `<RetVal>` indica il registro in cui è memorizzato il valore da restituire.

Costanti e tipi

- In alcune istruzioni che generano il codice è necessario indicare valori costanti e/o esplicitamente il tipo degli operandi.
- La gestione centralizzata (che evita duplicati) richiede che costanti e tipi siano definiti (e poi “recuperati”) nel contesto globale mantenuto da LLVM
- Il contesto è dunque sempre un parametro delle corrispondenti istruzioni
- Nel linguaggio sviluppato esiste solo il tipo `double`, per cui abbiamo utilizzato solo le due seguenti istruzioni
 - `ConstantFP::get(*<context>, APFloat(<costanteC++>))`
per creare (la prima volta) e poi recuperare la costante IR LLVM floating point corrispondente a `<costanteC++>`
 - `Type::getDoubleTy(*<context>)`
per specificare il tipo corrispondente a `double`

Gestione blocchi

- Per creare e posizionare i Basic Block e per stabilire il punto di inserimento del codice, abbiamo usato le seguenti istruzioni
 - `BasicBlock::Create(*<context>, <Label>[, <function>])`
che crea un Basic Block non ancora posizionato o (nel caso sia presente il terzo parametro) posizionato in fondo agli attuali blocchi della funzione `<function>`
 - `<function>->insert(<function>->end(), <BB>)`
che inserisce il Basic Block `<BB>` in coda ai blocchi che già concorrono alla definizione della funzione `<function>`
 - `<function>->getEntryBlock().begin()`
che definisce, come punto di inserimento del codice (da passare al builder), l'inizio del primo blocco (chiamato appunto *entry block*) della funzione `<function>`
 - `<builder>->GetInsertBlock()`
che recupera (il riferimento a) il Basic Block dove viene correntemente inserito il codice
 - `<builder>->SetInsertPoint(<BB>)`
che istruisce il builder ad inserire codice nel Basic Block `<BB>`

Operazioni a livello di modulo

- A livello di *modulo* che, ricordiamo, è il contenitore in cui sono inserite le definizioni di funzioni (ed altri “oggetti” di linkage globale), abbiamo utilizzato i seguenti metodi.
 - `module->getNamedGlobal(<Name>)`
che recupera (se presente nel modulo) la *variabile globale* identificata dal nome `<Name>`
 - `<module>->getFunction(<Name>)`
che recupera (se presente nel modulo) la funzione identificata dal nome `<Name>`
 - `new GlobalVariable(*<module>, <type>, false, <linkage>, <Val>, <Name>)`
Questa istruzione genera una variabile globale di tipo `<type>`, nome `<Name>` e valore iniziale `<Val>`. Il parametro `<linkage>` può essere a sua volta definito nel seguente modo
 - `GlobalValue::LinkageTypes <linkage> = GlobalValue::CommonLinkage`

Emissione del codice

- Dopo aver “preparato” il codice, l'emissione fisica in output (nel corso abbiamo sempre utilizzato `stderr`, seguendo il tutorial di Kaleidoscope) può avvenire in due modi differenti:
 - 1 Utilizzando “puntualmente” i metodi `print` di funzioni e variabili globali:
 - `<function>->print(errs())`
 - `<globalV>->print(errs())`
 - 2 Il secondo modo consiste nel “dump” completo del modulo (quindi senza le singole `print`, perché altrimenti si avrebbe doppia emissione):
 - `<module>->dump()`

Linguaggi e compilatori

1 Complementi

- Sul codice C++ usato per generare Human-Readable Intermediate Representation LLVM
 - Metodi di IRBuilder
 - Blocchi, moduli ed emissione del codice IR “human readable”
- Grammatica base per il progetto di primo livello

Grammatica primo livello

```
%start startsymb;

startsymb:
    program

program:
    %empty
|   top ";" program

top:
    %empty
|   definition
|   external
|   globalvar

definition:
    "def" proto block

external:
    "extern" proto

proto:
    "id" "(" idseq ")"
```

Implementare
questo
front-end

```
globalvar:
    "global" "id"

idseq:
    %empty
|   "id" idseq

%left ":";
%left "<" "==" ";
%left "+" "-";
%left "*" "/";

stmts:
    stmt
|   stmt ";" stmts

stmt:
    assignment
|   block
|   exp

assignment
    "id" "<=" exp
```

Grammatica primo livello

```
block:
    "{" stmts "}"
|    "{" vardefs ";" stmts "}"
```

```
vardefs:
    binding
|    vardefs ";" binding
```

```
binding:
    "var" "id" initexp
```

```
exp:
    exp "+" exp
|    exp "-" exp
|    exp "*" exp
|    exp "/" exp
|    idexp
|    "(" exp ")"
|    "number"
|    expif
```

```
initexp:
    %empty
|    "=" exp
```

```
expif:
    condexp "?" exp ":" exp
```

```
condexp:
    exp "<" exp
|    exp "==" exp
```

```
idexp:
    "id"
|    "id" "(" optexp ")"
```

```
optexp:
    %empty
|    explist
```

```
explist:
    exp
|    exp "," explist
```