



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

4. LAB 2: Introduzione ai *passi* di trasformazione LLVM

Linguaggi e Compilatori [I215-011]

Corso di Laurea in INFORMATICA
(D.M.270/04) [16-262]
Anno accademico 2023/2024

Prof. Andrea Marongiu
andrea.marongiu@unimore.it

Copyright note

È vietata la copia e la riproduzione dei contenuti e immagini in qualsiasi forma.

È inoltre vietata la redistribuzione e la pubblicazione dei contenuti e immagini non autorizzata espressamente dall'autore o dall'Università di Modena e Reggio Emilia.

Credits

- Pekhimenko, University of Toronto, “Compiler Optimization”

Ricorda: I *passi* LLVM

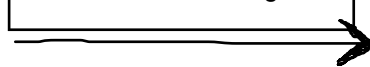
- Abbiamo visto che il middle-end è organizzato come una sequenza di *passi*
 - *Passi* di analisi
 - Consumano la IR e raccolgono informazioni sul programma
 - *Passi* di trasformazione
 - Trasformano il programma e producono nuova IR
- Ultimata la prima esercitazione, ci focalizziamo sui passi di trasformazione LLVM

La IR di LLVM

- La IR di LLVM utilizza la forma SSA (*Static Single Assignment*), per la quale una variabile non può essere definita più di una volta
- In questo modo i problemi che ci stiamo ponendo nelle lezioni teoriche appaiono non sussistere

```
main {  
    int a = 100;  
    int a = 42;  
    print a;  
}
```

Faccio in modo che ogni



```
main {  
    int a1 = 100;  
    int a2 = 42;  
    print a2;  
}
```

Come rilevare le *dead stores* nella DCE? (prossima lezione)

Nella forma SSA è triviale capire che la prima istruzione non è usata

La IR di LLVM

- La forma SSA è stata proposta come framework per semplificare l'ottimizzazione della IR a valle di tanti anni di ricerca sulla *dataflow* analysis
- È dunque da considerare un punto d'arrivo, e nelle lezioni teoriche capiamo come ci si è arrivati
- Per i laboratori sfruttiamo questo risultato come punto di partenza, e ci focalizziamo su come sfruttarlo per realizzare le nostre ottimizzazioni

Manipolazione delle istruzioni

Public Member Functions

	Instruction (const Instruction &)=delete
Instruction &	operator= (const Instruction &)=delete
Instruction *	user_back () Specialize the methods defined in Value , as we know that an instruction can only be used by other
const Instruction *	user_back () const
const BasicBlock *	getParent () const
BasicBlock *	getParent ()
const Module *	getModule () const Return the module owning the function this instruction belongs to or nullptr if the function does not
Module *	getModule ()
const Function *	getFunction () const Return the function this instruction belongs to.
Function *	getFunction ()
void	removeFromParent () This method unlinks 'this' from the containing basic block, but does not delete it.
SymbolTableList< Instruction >::iterator	eraseFromParent () This method unlinks 'this' from the containing basic block and deletes it.
void	insertBefore (Instruction * InsertPos) Insert an unlinked instruction into a basic block immediately before the specified instruction.
void	insertAfter (Instruction * InsertPos) Insert an unlinked instruction into a basic block immediately after the specified instruction.
SymbolTableList< Instruction >::iterator	insertInto (BasicBlock * ParentBB , SymbolTableList< Instruction >::iterator I) Inserts an unlinked instruction into ParentBB at position I and returns the iterator of the inserted
void	moveBefore (Instruction * MovePos) Unlink this instruction from its current basic block and insert it into the basic block that MovePos lives
void	moveBefore (BasicBlock & BB , SymbolTableList< Instruction >::iterator I) Unlink this instruction and insert into BB before I .
void	moveAfter (Instruction * MovePos) Unlink this instruction from its current basic block and insert it into the basic block that MovePos lives
bool	comesBefore (const Instruction * Other) const Given an instruction Other in the same basic block as this instruction, return true if this instruction c

- Ci sono molti modi (i.e., APIs) per manipolare le istruzioni
 - Instruction, BasicBlock,...
- Come sempre la documentazione ci aiuta a capire quali metodi (APIs) sono disponibili per una data classe

<https://llvm.org/doxygen/index.html>

User – Use - Value

- Supponiamo di dover ottimizzare il seguente codice:

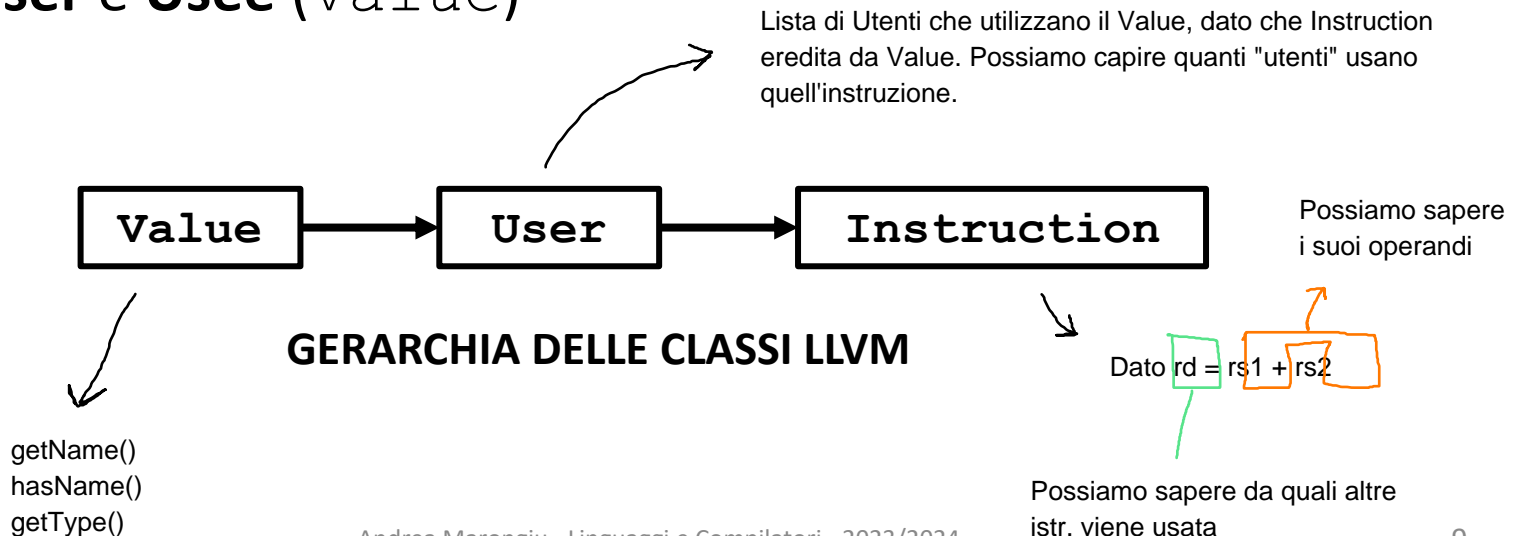
```
X %2 = add %1, 0           ; Identità algebrica  
  %3 = mul %2, 2
```



- È evidente che la prima istruzione contiene una *identità algebrica* e posso farne a meno
 - Ma cosa succede se semplicemente rimuovo l'istruzione?
 - Il programma va in *crash*, perché **non ho aggiornato le *references* correttamente**
- Come possiamo assicurarci che tutte le *references* (cioè gli *usi*) siano aggiornate correttamente?
- Sfruttando le relazioni ***User - Use - Value*** di LLVM

User – Use - Value

- Le istruzioni (`Instruction`) LLVM ereditano dalla classe `Value` (come quasi tutte le classi LLVM)
- Ma ereditano anche dalla classe `User`
 - Quindi esiste implicitamente un legame tra le **istruzioni** e i loro **usi**
- In altre parole, le `Instruction` giocano entrambi i ruoli di **User** e **Usee** (`Value`)



Value

- La classe `Value` è la più importante classe base in LLVM, dato che quasi tutti i tipi di oggetto ereditano da questa
- Un nodo `Value` ha un *tipo* (es., integer, floating point): **`getType()`**
- Un nodo `Value` può avere o meno un *nome*: **`hasName()`, `getName()`**
- Soprattutto, un nodo `Value` ha una lista di *Users* che lo utilizzano

Istruzioni come User

Un oggetto `Instruction` è anche un oggetto `User`

- Ogni `User (Instruction)` ha una lista di valori che sta utilizzando. Questi valori sono gli **operandi** dell'istruzione, e sono oggetti di tipo `Value`.

Stampo i suoi "usi" (operandi)

```
User &Inst = ...  
for (auto Iter = Inst.op_begin(); Iter != Inst.op_end(); ++Iter)  
    { Value *Operand = *Iter; }
```

Se eseguo questo codice per analizzare l'istruzione

`%2 = add %1, 0`

verranno estratti gli operandi



`%1, 0`

Istruzioni come Usee

Perché un oggetto `Instruction` è anche uno **Usee**?

- La risposta risiede nel come dobbiamo interpretare un oggetto `Instruction` LLVM:

```
%2 = add %1, 0
```

-  Il risultato dell'istruzione `add %, 0` viene assegnato a `%2`
-  `%2` è la rappresentazione `Value` dell'istruzione `add %1, 0`

- Quindi, ogni volta che nel testo usiamo il valore `%2` in realtà intendiamo proprio indicare l'istruzione `add %1, 0`

Ricapitolando...

Torniamo a considerare l'esempio di prima

```
%2 = add %1, 0           ; Identità algebrica  
%3 = mul %2, 2
```

Sia `Inst` una reference alla prima istruzione:

Da `User` l'istruzione usa degli operandi

```
for (auto Iter = Inst.op_begin(); Iter != Inst.op_end(); ++Iter)  
    { Value *Operand = *Iter; }
```

→ **Operand %1, 0**

Ma ha a sua volta degli users (ovvero, le istruzioni di cui è una usee)

```
for (auto Iter = Inst.user_begin(); Iter != Inst.user_end(); ++Iter)  
    { User *InstUser = *Iter; }
```

→ **Instruction mul %2, 2 (oppure Value %3)** → Poichè `<Value> = <Operand>`

Vediamo se abbiamo capito

Ipotizziamo di avere il seguente codice C

```
y = p + 1;  
y = q * 2;  
z = y + 3;
```

Cosa dovrebbe essere ritornato dalla routine che ispeziona gli *users* dell'istruzione $y = p + 1$?



Cioè user.begin() user.end()

Vediamo se abbiamo capito

Ipotizziamo di avere il seguente codice C

```
y = p + 1;  
y = q * 2;  
z = y + 3;
```

Cosa dovrebbe essere ritornato dalla routine che ispeziona gli *users* dell'istruzione $y = p + 1$?

- **Niente!**



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

LAB 2 – TRANSFORM PASS (LocalOpts)

LocalOpts – Sorgente

- Scaricate il sorgente per il Lab 2

`LocalOpts.cpp` (scheletro del passo)

`Foo.ll` (programma di test)

Esercizio 1 – Studio del passo

- Prima di tutto dobbiamo registrare il nostro passo LocalOpts come abbiamo imparato a fare
 - Copiando dove opportuno LocalOpts.cpp (SRC_ROOT/lib/Transforms/Utils)
 - Aggiungendo LocalOpts.cpp al file CMakeLists.txt nella stessa cartella
 - Creando il file LocalOpts.h (SRC_ROOT/include/llvm/Transforms/Utils)
 - Editando PassRegistry.def e PassBuilder.cpp (SRC_ROOT/lib/Passes)
- A questo punto possiamo ricompilare e installare opt
- Essendo un passo di trasformazione la IR verrà modificata, e quindi dobbiamo specificare l'output file

```
opt -p localopts Foo.ll -o Foo.optimized.bc
```

- L'output è in formato bytecode, quindi dobbiamo usare il disassembler per generare la forma .ll leggibile

```
llvm-dis Foo.optimized.bc -o Foo.optimized.ll
```

Esercizio 1 – Studio del passo

- Confrontare `Foo.ll` e `Foo.optimized.ll`, e tramite lo studio del passo in `Transform.cpp` capire cosa fa (e come lo fa)
- Familiarizzare un po' con le varie primitive di manipolazione della IR proposte nel passo
- Studiare la documentazione, rispondere alla domanda presente nel commento verso la fine del passo

`Transform.cpp`



`LocalOpts.cpp`

Esercizio 2 – Un passo più utile

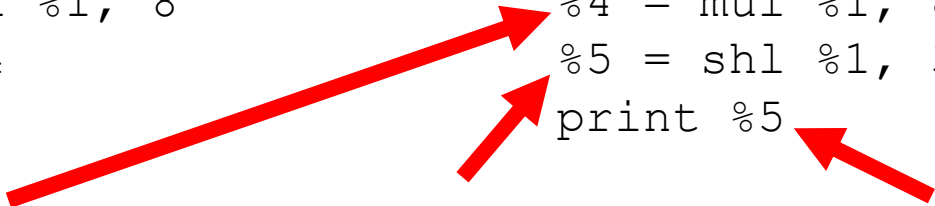
- Per esercitarci a manipolare la IR modifichiamo adesso il passo Transform.cpp perché sostituisca tutte le operazioni di **moltiplicazione** che hanno tra gli operandi una costante che è una potenza di 2 con una **shift** (*strength reduction*)

Es.

```
%1 = add %2, %3  
%4 = mul %1, 8  
print %4
```

DIVENTA

```
%1 = add %2, %3  
%4 = mul %1, 8  
%5 = shl %1, 3  
print %5
```



NOTA: Non rimuoviamo la **mul**, ma la slinkiamo dal programma rimpiazzando i suoi usi con gli usi della nuova istruzione