Italian C++ Community

++it

www.italiancpp.org

All’interno di Clang e LLVM

Concetti, design e implementazione

Di cosa parleremo

- LLVM e Clang
  - Struttura del progetto
  - Architettura generale
- Clang: design e implementazione
  - Lexer
  - Parser
  - Struttura dell'AST
  - Manipolazione dell'AST
  - Analisi semantica
  - Generazione del codice intermedio
- Varie ed eventuali

Il progetto LLVM

Cos'è LLVM?

- Un progetto ad ampio spettro che fornisce una vasta gamma di strumenti per la realizzazione di compilatori ed interpreti
- Insieme di libreria per la generazione, manipolazione e *ottimizzazione* del *LLVM Intermediate Representation* (IR)
- Copre tutta l'infrastruttura, dall'IR in poi: debugger, linker, binutils, interprete con Just-In-Time compilation, ecc…

Il progetto LLVM

Viene usato come backend per un gran numero di linguaggi sia interpretati sia compilati, e di progetti correlati. Ad esempio:

Linguaggio	Progetto e/o Produttore
C, C++ e Objective-C	Clang
C++ → asm.js	Emscripten/Mozilla
Interprete C++	Root/CERN
Haskell	GHC
Rust	Mozilla
JavaScript	WebKit
OpenCL e GLSL	Intel, nVidia, Apple
.NET IL	Mono
Python	PyPy

Clang

Cos'è Clang:

- Parte integrante del progetto LLVM
- Un compilatore per i linguaggi C, C++ e Objective-C
- Un ampio insieme di librerie per leggere, manipolare, analizzare, generare codice scritto in questi linguaggi

Clang

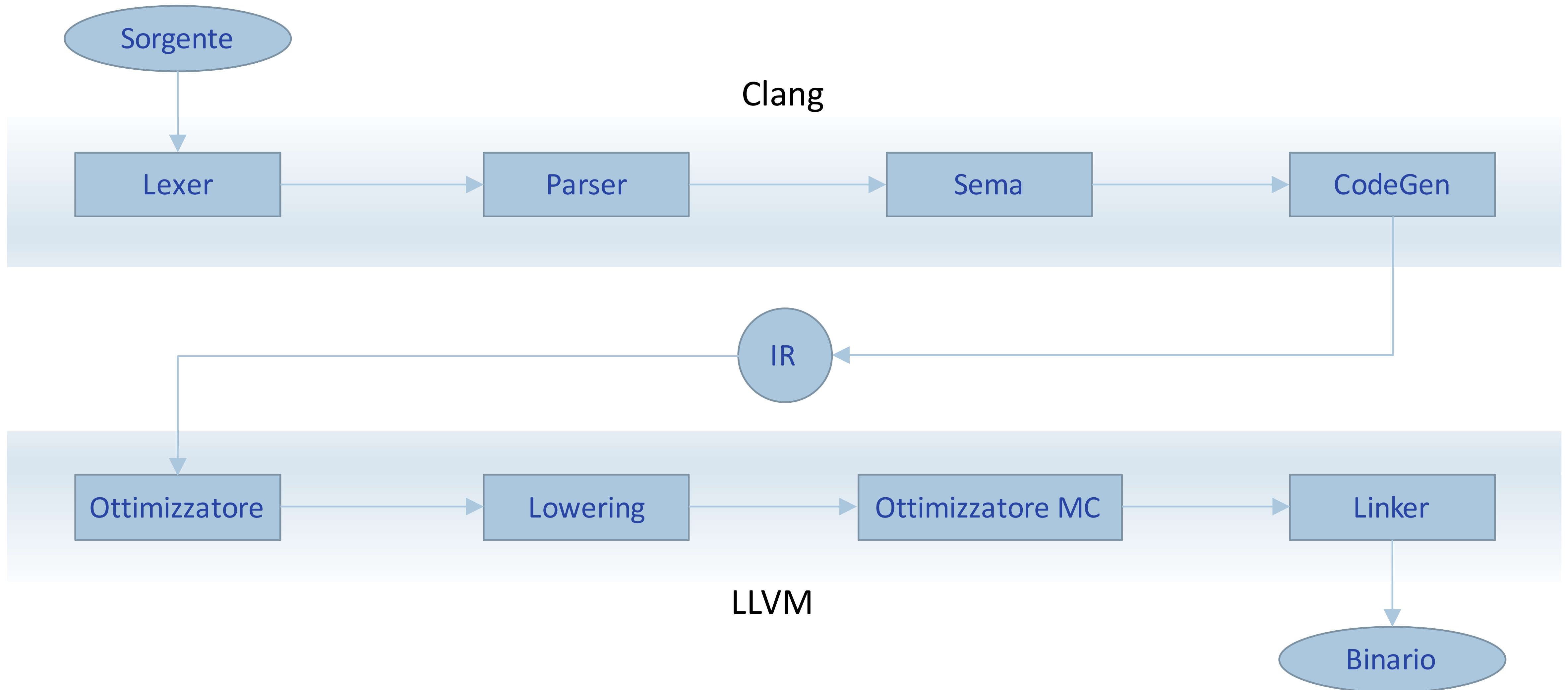
Goal del progetto:

- Bassi tempi di compilazione
- Ottimizzazione del codice generato
- Messaggi di errore chiari e ben strutturati
- Eccellente aderenza agli standard:
  - C89, C99, C11
  - C++98/03, C++11, C++14
  - OpenCL
- Integrazione con tool e IDE:
  - Code completion, syntax highlighting, indexing, refactoring, ecc…
- Compatibilità *drop-in* con GCC (e VC++, sperimentale)

Storia del progetto

- 2002 – Chris Lattner pubblica la sua tesi di dottorato allo UCLA:
  LLVM: An Infrastructure for Multi-Stage Optimization
- *2005* – Apple adotta il progetto e incorpora LLVM nello stack OpenGL di OS X 10.4
- 2007 – Nasce il progetto Clang
- 2010 – Clang 2.7 supporta il C++ tanto da riuscire a compilare se stesso
- 2010 – Nasce il progetto `libc++` per reimplementare una libreria standard C++11
- 2012 – Clang 3.3 supporta completamente il C++11
- 2014 – Clang 3.5 supporta il C++14 prima che venga standardizzato 😎
- 2015 – Microsoft annuncia l'inclusione di Clang in Visual Studio

Architettura generale



Backend

LLVM si occupa della parte di *backend:*

- Definisce un linguaggio di *rappresentazione intermedia*
- Fornisce una serie di tool per creare, manipolare, analizzare programmi scritti in IR
- Il codice intermedio viene analizzato e manipolato da una serie di *passi*, ad esempio:
  - Dead Code Elimination, Constant Propagation, Alias Analysis, Tail Call Elimination
- Ogni passo modifica localmente l'IR e/o estrae informazioni utili ai passi successivi
- La fase di lowering traduce l'IR in assembly specifico per l'architettura selezionata

Codice intermedio di LLVM

Due parole sull'IR:

- Codice RISC-like di basso livello ma target-independent
- Ogni operazione osserva un semplice ma rigido type-system
- Notazione esplicita per chiamate a funzione, gestione delle eccezioni, aritmetica dei puntatori, istruzioni vettoriali, ecc…

Frontend

In questo talk ci concentreremo su Clang, il frontend:

- Architettura generale
- Design delle strutture dati utilizzate
- Problematiche specifiche del C++: istanziazione dei template, valutazione funzioni constexpr, gestione delle feature supportate dai diversi standard implementati, ecc.…
- Interfaccia con l’esterno
- Uno sguardo al codice

Struttura del progetto

Clang è formato da un insieme di componenti indipendenti, tra cui:

- `libBasic` – Utility generali, descrizione dei target, gestione dell'I/O, infrastruttura per le diagnostiche
- `libLexer` – Implementa il preprocessore e il Lexer
- `libAST` – Dichiara la struttura dati dell'*Abstract Syntax Tree*
- `libParser` – Produce l'AST a partire dal flusso di token generato dal Lexer
- `libSema` – Effettua l'analisi semantica del codice: type-check, lookup dei nomi, istanziazione dei template, ecc…
- `libCodeGen` – Genera il *Control Flow Graph* a partire dall'AST ed emette il codice intermedio LLVM
- `clang` – Il compilatore vero e proprio, utilizzabile da terminale

Il Lexer

Il modulo Lexer di Clang implementa in un singolo passo le fasi di *preprocessing* e *analisi lessicale* del codice:

- Il testo sorgente viene trasformato in una sequenza di *token*
- Il Lexer stesso si occupa della gestione delle *macro*
- La classe `SourceManager`, per ogni `Token`, tiene traccia di:
  - Di che tipo di token si tratta
  - La posizione del token all'interno del file
  - La provenienza del token nello stack di espansione delle macro

Il Lexer

Risultato:

```
prova.cpp:8:2: error: assigning to 'int *'
                     from incompatible type 'int'
 THREE(p);
 ~~~~~~~~
prova.cpp:3:18: note: expanded from macro 'THREE'
#define THREE(x) TWO(x)
                 ~~~~~~
prova.cpp:2:16: note: expanded from macro 'TWO'
#define TWO(x) ONE(x)
               ~~~~~~
prova.cpp:1:18: note: expanded from macro 'ONE'
#define ONE(x) x = 1
                 ^ ~
```

Hands on

Il Parser

Il Parser di Clang è scritto a mano, non si utilizzano tool di generazione automatica come Yacc o Bison:

- Generazione delle diagnostiche molto più pulita e precisa
- Maggiore controllo sulle performance del codice
- Un parser unico deve supportare una grande varietà di linguaggi diversi:
  - C, C++, Objective-C, OpenCL, ecc... in una varietà di versioni ed estensioni diverse
  - Impossibile da mantenere in un parser generato in automatico
- La generazione dell'AST da parte del Parser è intervallata alla sua analisi da parte di `libSema`:
  - Parsing e analisi avvengono in un'unica passata
  - Non viene mai prodotto un AST "parziale"

Struttura dell’AST

L’Abstract Syntax Tree è la struttura dati che rappresenta la forma sintattica del codice.

Il design dell’AST di Clang è influenzato da vari requisiti:

- Efficienza e uso di memoria
- Possibilità di generare diagnostiche chiare e precise
- Alta fedeltà al codice originario per i casi d’uso diversi dalla semplice compilazione:
  - Indexing
  - Code completion
  - Refactoring
  - …

Struttura dell'AST

Alcune scelte di design seguono da quanto detto:

- Ogni nodo mantiene traccia della precisa posizione dei token da cui proviene
- L'AST non è solo sintattico, ma contiene tutte le informazioni semantiche estratte da `libSema`:
  - Tipo delle espressioni
  - Posizione delle conversioni di tipo implicite
  - L'AST della dichiarazione dei template e di ogni singola istanza
- Si sfrutta il piu` possibile lo sharing di sottoalberi in comune (complica la gestione della memoria? tl;dr: no)

Contesto di parsing

Tutti gli elementi dell’AST vivono sotto il controllo di un oggetto di tipo `ASTContext`, che ne è il proprietario:

- Un `ASTContext` gestisce la compilazione di una singola compilation unit
- Gestisce l’allocazione della memoria per tutti i nodi dell’AST:
  - Qualsiasi nodo dell’AST viene creato tramite l’`ASTContext`
  - L’`ASTContext` si preoccupa di verificarne l’unicità (per i tipi di nodi che devono essere unici)
  - Al termine della compilazione l’`ASTContext` distrugge tutto l’AST in un colpo solo
- Vantaggi:
  - Gestione della memoria fortemente semplificata: uso libero di puntatori grezzi
  - Allocazione contigua di oggetti dell’AST che rappresentano pezzi di codice vicini
- L’oggetto ASTContext è serializzabile

Gestione dei tipi

In Clang ogni tipo è rappresentato da un oggetto di una sottoclasse di `Type`:

- I tipi sono "unici": ogni riferimento a "int" è rappresentato dallo stesso oggetto di tipo `IntTy` (limitatamente ad un singolo `ASTContext`)
- Gli oggetti Type sono associati a tipi *non qualificati*
- Il tipo QualType rappresenta un tipo con dei qualificatori associati (`const`, `volatile`, …)
- La presenza dei qualificatori è mantenuta da QualType nei bit meno significativi del puntatore al Type:
  - L'uguaglianza di tipi resta un confronto tra due puntatori
  - L'utilizzo di memoria resta quello di un singolo puntatore
  - Operazione trasparente nascosta dall'interfaccia di QualType (che è un regular type)

Gestione dei tipi

È necessario tenere traccia di come i tipi vengono effettivamente indicati dal programmatore nel codice sorgente:

- Sappiamo che il tipo di una variabile è T, ma:
  - Il programmatore ha scritto proprio T?
  - Ha usato un typedef?
  - Il typedef era in scope o raggiunto tramite un qualificatore (es. a::b::type)?
  - Dove sono dichiarati i tipi e/o i typedef usati?
- Typedef, elaborated types, ecc… sono indicati da oggetti Type appositi.
- Ogni Type ha il proprio “canonical type”, che punta al vero tipo.

Consumare l'AST

L'AST di Clang è pensato per essere utilizzato da vari tipi di tool di sviluppo per ottenere informazioni sul sorgente:

- Per esigenze comuni, rivolgersi alle librerie ausiliarie dedicate:
  - libTooling
  - libFormat
  - libEdit
  - libRewrite
- L'entry point per visitare direttamente l'AST è la classe `ASTConsumer`:
  - Ogni "client" dell'AST deve essere un consumer
  - Interfaccia in sitle SAX: il consumer riceve parti dell'AST mentre vengono trovate
- La navigazione dell'albero è invece delegate agli `ASTMatcher`:
  - Interfaccia in stile "pattern matching" per scendere nell'albero in modo dichiarativo

AST Matchers

Interfaccia dichiarativa per visitare l'AST ed estrarre i nodi di interesse.

Esempio:

```
recordDecl(hasDescendant(
           ifStmt(hasThen(
                  stmt(hasDescendant(
                       ifStmt())))))))
```

Questa espressione identifica qualunque nodo che:

- È la dichiarazione di un "record" (struct/class/union) che…
- …ha come discendente qualsiasi un'istruzione "if"…
- …il cui blocco "then" è un'istruzione che …
- …tra i discendenti contiene un'altra istruzione "if"

```
class C {
  void func(int x, int y) {
    if(x > 0) {
      while(true) {
        if(y > 0) break;
      }
    }
  }
};
```

Hands on

Analisi semantica

- Il modulo `libSema` si occupa della fase semantica della compilazione:
  - Lookup dei nomi
  - Overload resolution
  - Type-checking
  - Istanziazione dei template
  - …
- I controlli flow-dependent (es. no return alla fine di una funzione) vengono eseguiti creando un *Control Flow Graph*:
  - Grafo orientato di blocchi di istruzioni di base
  - Un arco ad ogni salto
  - Rappresentazione esplicita del flusso di controllo implicito come costruttori, distruttori, eccezioni, ecc…
  - Struttura condivisa con `libAnalysis` (che implementa lo Static Analyzer)

Templates

La gestione dei template è uno dei punti più delicati di un compilatore C++.

L'implementazione di Clang segue le due fasi già definite dallo standard:

- Il template viene parsato nel momento in cui viene dichiarato, e si produce (ad esempio) una dichiarazione di tipo `ClassTemplateDecl`
- Ogni tipo type-dependent viene rappresentato da oggetti di tipo `TemplateTypeParmType` e altri (es. `DependentDecltypeType`)
- L'istanziazione viene eseguita da una sottoclasse di `TreeTransform`:
  - Classe astratta (CRTP, in realtà) per l'implementazione di trasformazioni dell'AST
  - L'AST della dichiarazione del template viene traformato sostituendo i parametri
  - Il risultato diventa una dichiarazione `ClassTemplateSpecializationDecl` (o simile per le funzioni) e aggiunto all'AST assieme al resto

Generazione del codice

Il codice intermedio LLVM viene generato dal modulo `libCodeGen`:

- L’AST viene visitato in modo depth-first per generare il codice intermedio
- In questo punto viene gestita la ABI:
  - layout
  - gestione delle eccezioni
  - name mangling
  - …
- Non si ripetono features fornite già dal backend:
  - Quasi niente constant folding
  - Il codice è generato nel modo più semplice possibile, il backend poi ottimizzerà
  - Il calcolo degli offset e dell’aritmetica dei puntatori è delegato alle apposite istruzioni IR

Hands on

Cosa resta

Cosa manca al nostro tour:

- I restanti dettagli del processo di compilazione:
  - libFrontend – API per gestire ad alto livello le fasi della compilazione
  - libDriver – Gestione della toolchain, gestione della cross-compilazione, invocazione dei sottoprocessi, …
- Librerie per l'analisi e la modifica diretta del sorgente
  - libTooling – Framework generale per la realizzazione di tool basati su Clang
  - libRewrite – Refactoring
  - libFormat – Riformattazione del codice secondo determinate linee guida
  - libclang – API C stabile verso il parser e l'AST (utile per bindings ad altri linguaggi)

Domande?