Compiler Construction & Tarjan's Algorithm

Understanding SCCs in Call Graph Analysis

December 1, 2025

New Paltz
STATE UNIVERSITY OF NEW YORK

Table of Contents

Overview

This presentation covers:

- Fundamentals of Compiler Construction
- Graph Theory in Compilers
- Real-World Application: Call Graph Analysis
- LLVM Integration

Table of Contents

What is a Compiler?

Definition

A **compiler** is a program that translates source code from a high-level programming language into machine code or intermediate representation.

What is a Compiler?

Definition

A **compiler** is a program that translates source code from a high-level programming language into machine code or intermediate representation.

Key Responsibilities

- Translate human-readable code to executable format
- Check for syntax and semantic errors
- Optimize code for performance
- Generate warnings and error messages

Compiler Phases

The compilation process consists of several phases:

1. **Lexical Analysis** - Breaking source into tokens
2. **Syntax Analysis** - Building parse trees
3. **Semantic Analysis** - Type checking, scope resolution
4. **Intermediate Code Generation**
5. **Optimization**
6. **Code Generation**

Frontend vs Backend

Frontend

- Lexical Analysis
- Parsing
- Semantic Analysis
- Symbol Tables
- Type Checking

Backend

- Code Optimization
- Register Allocation
- Code Generation
- Target-specific transforms

Semantic Analysis Phase

- **Type Checking**: Verify type compatibility
- **Scope Resolution**: Variable and function visibility
- **Flow Analysis**: Control and data flow
- **Call Graph Construction**: Function dependencies

Semantic Analysis Phase

- **Type Checking**: Verify type compatibility
- **Scope Resolution**: Variable and function visibility
- **Flow Analysis**: Control and data flow
- **Call Graph Construction**: Function dependencies

Why Call Graphs Matter

Call graphs help detect:

- Recursive functions (direct and indirect)
- Unreachable code
- Optimization opportunities
- Stack overflow risks

Table of Contents

Call Graphs

Definition

A **call graph** is a directed graph where nodes represent functions and edges represent function calls.

Call Graphs

Definition

A **call graph** is a directed graph where nodes represent functions and edges represent function calls.

Example

If function `f()` calls `g()`, we draw an edge: `f` $\rightarrow$ `g`

Call Graphs

Definition

A **call graph** is a directed graph where nodes represent functions and edges represent function calls.

Example

If function `f()` calls `g()`, we draw an edge: $\mathtt{f} \rightarrow \mathtt{g}$

Purpose

- Analyze program structure
- Detect recursion (cycles in the graph)
- Optimize function inlining
- Determine call order

Table of Contents

nicoLang Compiler

Project Overview

nicoLang is a compiler project that implements call graph analysis using Tarjan's algorithm to detect recursive functions.

- Written in C++
- Header-only implementation
- Detects both direct and mutual recursion
- Provides detailed SCC reporting

Usage in Compilation

1. During semantic analysis, record function calls
2. Build the call graph incrementally
3. After parsing, run `analyze()`
4. Tarjan's algorithm detects all SCCs
5. Compiler can then:
    - Warn about recursion
    - Disable certain optimizations
    - Verify tail-call optimization applicability
    - Calculate stack depth requirements

Why This Matters

- **Performance**: $O(V + E)$ linear time complexity
- **Correctness**: Detects all forms of recursion
- **Optimization**: Enables better code generation
- **Error Prevention**: Warns about stack overflow

Why This Matters

- **Performance**: $O(V + E)$ linear time complexity
- **Correctness**: Detects all forms of recursion
- **Optimization**: Enables better code generation
- **Error Prevention**: Warns about stack overflow

Real-World Impact

- Used in GCC, LLVM, and other major compilers
- Essential for functional programming languages
- Critical for optimization passes

Table of Contents

What is LLVM?

LLVM Overview

LLVM (Low Level Virtual Machine) is a modern compiler infrastructure that provides:

- Intermediate Representation (IR) for code optimization
- Modular compiler architecture
- Language-independent optimization framework
- Used by Clang, Swift, Rust, and many others

What is LLVM?

LLVM Overview

LLVM (Low Level Virtual Machine) is a modern compiler infrastructure that provides:

- Intermediate Representation (IR) for code optimization
- Modular compiler architecture
- Language-independent optimization framework
- Used by Clang, Swift, Rust, and many others

- **Frontend**: Language-specific parsing to LLVM IR
- **Middle-end**: Optimization passes on IR
- **Backend**: IR to machine code generation

LLVM Optimization Passes

Call Graph-Based Optimizations

LLVM uses call graph analysis for multiple optimization passes:

- **Function Inlining**: Inline small, frequently-called functions
- **Dead Argument Elimination**: Remove unused function parameters
- **Interprocedural Constant Propagation**: Propagate constants across function boundaries
- **Tail Call Optimization**: Convert recursive calls to iterations

LLVM Optimization Passes

Call Graph-Based Optimizations

LLVM uses call graph analysis for multiple optimization passes:

- **Function Inlining**: Inline small, frequently-called functions
- **Dead Argument Elimination**: Remove unused function parameters
- **Interprocedural Constant Propagation**: Propagate constants across function boundaries
- **Tail Call Optimization**: Convert recursive calls to iterations

Bottom-Up Processing

SCCs enable bottom-up analysis: analyze callees before callers, maximizing optimization opportunities.

Using LLVM in nicoLang

Integration Approach

We leverage LLVM's optimization infrastructure for call graph analysis and optimization passes.

- Generate LLVM IR from nicoLang source code
- Use optimization passes for code improvement
- Detect recursive functions through SCC analysis
- Generate optimized machine code

Using LLVM in nicoLang

Integration Approach

We leverage LLVM's optimization infrastructure for call graph analysis and optimization passes.

- Generate LLVM IR from nicoLang source code
- Use optimization passes for code improvement
- Detect recursive functions through SCC analysis
- Generate optimized machine code

Benefits

LLVM provides production-ready infrastructure for sophisticated compiler optimizations and analysis.

Table of Contents

Summary

- Compilers transform source code through multiple phases
- Call graphs represent function dependencies
- SCCs identify recursive function groups
- Tarjan's algorithm efficiently finds all SCCs
- nicoLang implements this for recursion detection

Key Takeaway

Graph algorithms like Tarjan's SCC are fundamental to modern compiler construction, enabling sophisticated analysis and optimization.

References

- Robert Tarjan: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing, 1(2):146-160, 1972.
- LLVM Project: The LLVM Compiler Infrastructure. https://llvm.org/

Thank You!
Questions?