City University of New York (CUNY)
CUNY Academic Works

Open Educational Resources | Hunter College

2023

Introduction

Raffi T. Khatchadourian
CUNY Hunter College

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/hc_oers/42
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Programming Languages/Software Engineering

Introduction

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Programming Languages, Software Engineering, and Software Evolution

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Programming Languages, Software Engineering, and Software Evolution

- Change is at the heart of software development.
- To change software, developers need to understand complex relationships between programing language constructs (Object-Oriented, Dynamic Languages).
- Developers must also engineer their changes to meet their goals, maintain architectural integrity, and not cause *regressions*.

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Research in Software Evolution

- How to ease the burden of efficiently, correctly, and securely evolve and maintain large and complex software systems?
- What are the common changes in software in practice? How can we *analyze* them? Leverage them?
- What are the relationships between those changes and resolving *technical debt*?
- How can we *automate* software evolution and maintenance?
- How can we *infer* safe and effective program transformations?
- How can we analyze software before and after transformations *without* running the code?

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Automated Evolution Examples

- Migrate legacy systems to utilize new programming language constructs.
- Upgrading library/framework APIs.
- Converting sequential code to parallel (and vice-versa) for enhanced efficiency?
- Improve responsiveness (e.g., sync -> async).

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Quiz 1: About You

1. Name?
2. Occupation, recreating, motivation?
3. Grad program (e.g., CS PhD, MS/MA), years of study, who is your grad advisor?
4. Background (e.g., undergrad PL/SE courses you've taken, industrial PL/SE experience, other CS background—such as ML).
5. One thing you expect to take out of this course?
6. What are your post graduation plans?

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Topics in Software Maintenance and Evolution

- Reverse engineering and re-engineering.
- Software refactoring and restructuring.
- Software migration, renovation, and rejuvenation.
- Software and system comprehension.
- Software repository analysis and mining.
- Code cloning and provenance.
- Concept and feature location.
- Change and detect management.
- Evolution of non-code artifacts.
- Software testing.
- Maintenance and evolution processes.
- Software quality assessment.
- Run-time evolution and dynamic configuration.
- Human aspects of software evolution.

Theme:

ML/DL programming and software evolution

- Machine Learning (ML)/Deep Learning (DL) becoming pervasive.
- Mission-critical applications (e.g., autonomous cars).
- Run-time performance (a lot of data, especially for DL).
- Programming is close to the hardware but fast.
- Programming is difficult to get speed (especially for training).
- Easier programming sacrifices speed (run-time efficiency).
- How to get the best of both worlds for *legacy* DL systems?

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Course Logistics

- Review syllabus.
- Get familiar with Bb page.
- Bb discussion board.

Research-Oriented Course

- Learn fundamentals but with an eye towards *research*!
- At times, the course may feel “unorganized.”
- Many choice, need to choose.
- Course structure is fixed, but the content is “dynamic.”
- Complete a research or industrial-novel project of your choice (individual or teams of 2-3 students).
    - Follow the steps of open-ended/risky research (proposal, fit in PL/SE literature, evaluate empirically or by mathematical proof).
    - At the end, you will have produced a research paper that you can submit to a conference.
    - Why? Equips you to conduct novel R&D.

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Research-Oriented Course

- Participate in class discussions and activities.
- Read research papers.
  - Later on, papers that match your project topic.
- Paper critiques.
  - Submit before class.
  - Why? Equips you to think critically.
- Research presentation.
  - You prepare and deliver for selected research papers.
  - Why? Equips you to communicate your ideas.
- Research projects (**not** implementing an "app")
  - Individual or teams of 2-3.

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

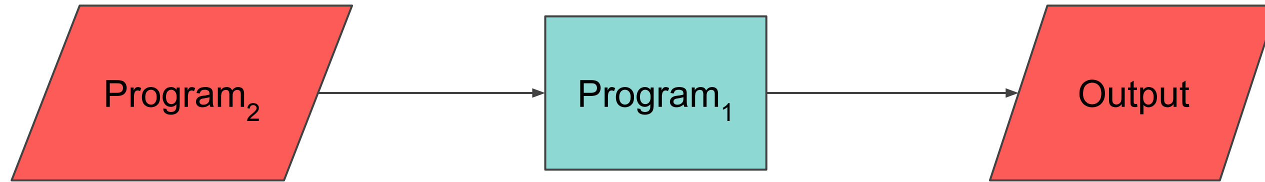
Introduction to Program Analysis and Transformation

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

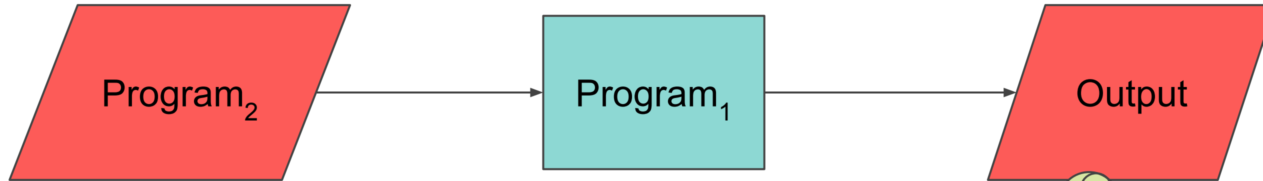
Traditional Programs

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Programs Analysis

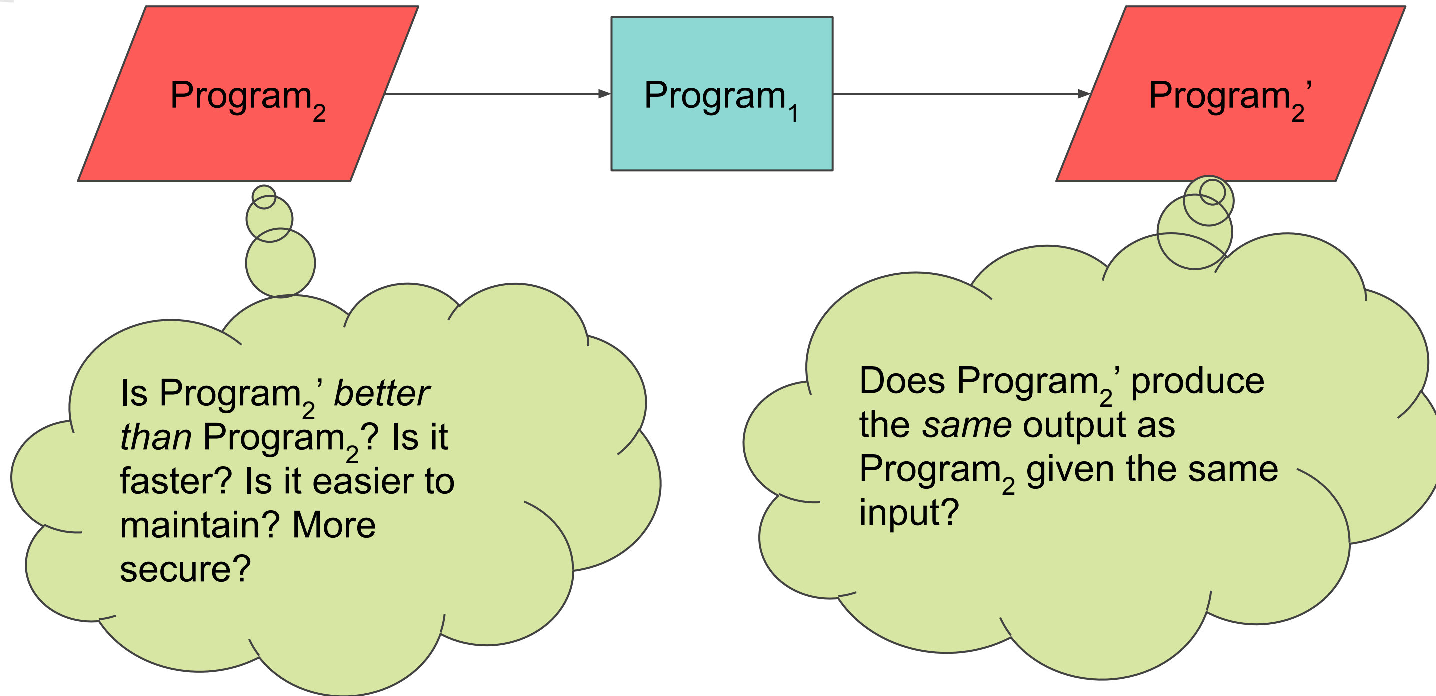


Programs Analysis



- How many variables?
- How many functions?
- Is the architecture reusable?
- What is the test *coverage?*
- Are bugs likely?
- Are there *security* vulnerabilities?

Programs Transformation (Refactoring)



Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

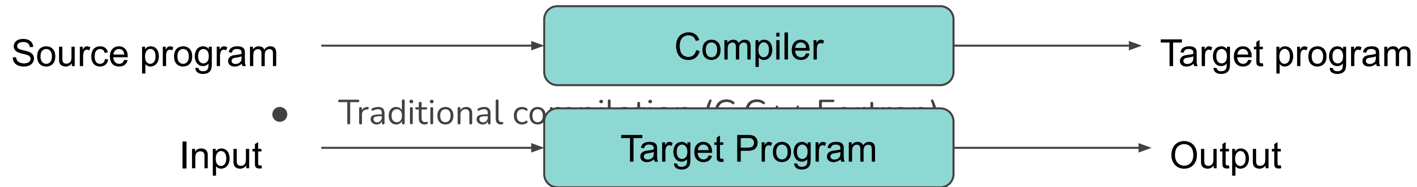
Research Topics

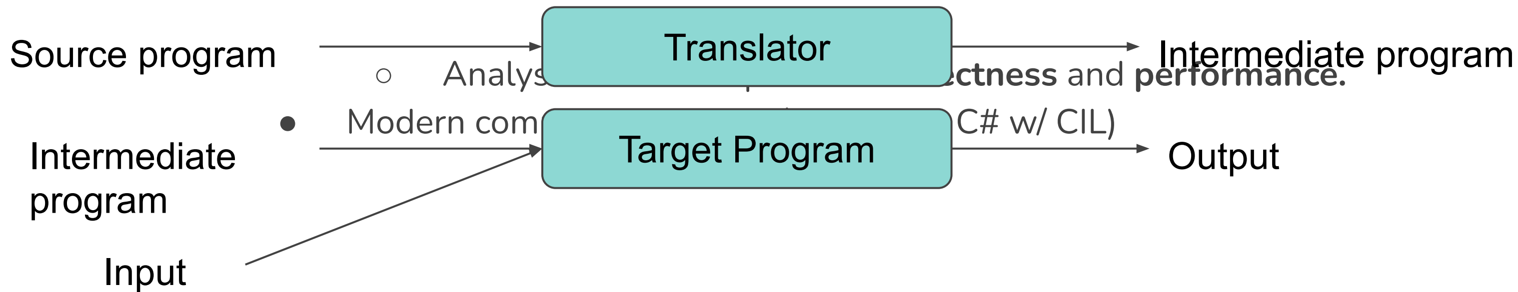
- Study how programmers code.
- Study how programmers *change* code.
- How to deal with large and complex software?
- How to locate potential *performance* improvements?
- How to locate potential *bugs*?
- How to discover potential security flaws?
- Can we *automatically* change programs to use latest language and platform features?
- How do we ensure that the new program *works* the same as the old one?
- Is the new program *designed* better than the old one? Is it easier to *maintain*?

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Use Cases for Compile-Time Analysis

- Traditional co
- ○ Analys ectness and **performance.**
- Modern com C# w/ CIL)





Use Cases for Compile-Time Analysis

- Software development environments
  - E.g., in Eclipse: finds **code smells** and potential **defects**; performs **code refactoring**.
- Software verification/checking tools
  - Prove the absence of certain categories of **defects**.
- Testing tools
  - E.g., for regression testing – which tests do **not** need to be rerun after some changes to the program?
- Also: comprehension tools, debugging tools (after failure), performance analysis tools, etc.
- More generally, **static analysis** (vs. **dynamic analysis**).

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Inside a Traditional Compiler: Front End

- Lexical analyzer (aka scanner)
  - Converts ASCII or Unicode to a **stream of tokens**.
- Syntax analyzer (aka parser)
  - Creates a **parse tree** from the token stream.
- Semantic analyzer
  - Type checking and conversions; other semantic checks
  - **Some compile-time analyses done here, on the AST**
- Generator of intermediate code
  - A parse tree is too high-level for code generation & optimization
  - Create **lower-level intermediate representation (IR)**, e.g., three-address code.

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Inside the Compiler: Middle Part

- **Compile-time analysis of intermediate code**
    - Additional IRs: control-flow graph (CFG), static single assignment form (SSA), def-use graph, etc.
    - Control-flow analysis, data-flow analysis, pointer analysis, side-effect analysis, polyhedral analysis,
- Machine-independent **optimization** of intermediate code: better three-address code.
    - Copy propagation, dead code elimination, code motion, constant propagation, redundancy elimination, parallelization, data locality optimizations, ...
- Currently, this is where most of **compiler research** is focused.

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Three-Address Code

- ASTs are high-level IRs
  - Close to the source language.
  - Suitable for tasks such as type checking and type inferencing.
- Three-address code is a lower-level IR
  - Closer to the target language (i.e., assembly code)
  - Suitable for tasks such as code generation/optimization
- Basic ideas
  - A small number of simple instructions: e.g. `x = y op z`
  - A number of compiler-generated temporary variables
    `a = b + c + d;` in source code `t = b + c; a = t + d;`
  - Simple flow of control – conditional and unconditional jumps to labeled statements

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Important Note

- The choice of the program representation on which to perform analysis is **critical**
  - E.g. if you are writing an Eclipse plug-in, you have access to the AST, but not to a lower-level IR
    - Plus, the results of the analysis are useful for ASTs (e.g., code smells reported to the programmer)
- In a compiler, we usually prefer to have access to a lower-level IR, since the analyses and transformations are easier
- In this course, we will focus on this scenario.

Addresses and Instructions

- “Address”: a program variable, a constant, or a compiler-generated temporary variable
- Instructions
  - `x = y op z`: binary operator op; y and z are variables, temporaries, or constants; x is a variable or a temporary
  - `x = op y`: unary operator op; y is a variable, a temporary, or a constant; x is a variable or a temporary
  - `x = y`: copy instruction; y is a variable, a temporary, or a constant; x is a variable or a temporary
  - Arrays, flow-of-control
  - Each instruction contains at most three “addresses”
    - Thus, **three-address code**

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Simple Examples

`x = y` produces one three-address instruction

Left: a pointer to the symbol table entry for x
Right: a pointer to the symbol table entry for y
For convenience, we will write this as `x = y`

`x = - y` produces `t1 = - y; x = t1;`

`x = y + z` produces `t1 = y + z; x = t1;`

`x = y + z + w` produces `t1 = y + z; t2 = t1 + w; x = t2;`

`x = y + - z` produces `t1 = - z; t2 = y + t1; x = t2;`

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

Flow of Control

- Three-address instructions
  - `goto L`: unconditional jump to the three-address instruction with label L
  - `if (x relop y) goto L`: x and y are variables, temporaries, or constants; relop ∈ { <, <=, ==, !=, >, >= }
- The labels are symbolic names

Credits: Slide content may contain repurposed material originally by Danny Dig and Atanas Rountev.

More Examples

- Possible three-address code: two versions
    - Example: **`if (x < 100 || x > 200 && x != y) x = 0;`**

```
if (x < 100) goto L2;
goto L3;
L3: if (x > 200) goto L4;
goto L1;
L4: if (x != y) goto L2;
goto L1;
L2: x = 0;
L1: …;
```

```
if (x < 100) goto L2;
if (x <= 200) goto L1;
if (x == y) goto L1;
L2: x = 0;
L1: …;
```

Main Topics

- **Control-flow analysis**: what **sequences of instructions** could be executed at run time?
  - Infinite number of sequences -> need finite static representation (control-flow graph)
- **Data-flow analysis**: what are the effects of these instruction sequences on the **state of the program**?
  - Infinite (or very large) sets of possible states -> need finite/small abstractions, often with loss of precision
- Key technical challenges: abstractions must be
  - **correct** (depending on the client)
  - **precise** and **efficient-to-compute**
- **Code transformations**: enabled by analysis.
  - In refactoring, this is called **precondition checking**.