The CAT team

Simone Campanoni

Umang Dhandhania
What we are going to do

• Teach you **code analysis and transformation**

- What they do
- What they could do
- What they can’t do
Who you are

• An engineer

• A C++ developer
  (you don’t have to be an incredible coder)

• An enthusiastic learner
Software knowledge assumed

• You know how to write C++ code in Linux platforms (e.g., class, inheritance, method overloading, containers like a set)
  C++ tutorial: http://www.cplusplus.com/doc/tutorial/

• You know Makefile
  Makefile tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

• You know how to debug C++ code
  gdb tutorial: https://www.tutorialspoint.com/gnu_debugger/index.htm
Machines to use for this class

You have access to the following machines, which are used to test your homework

• Wilkinson lab
  gotham.ece.northwestern.edu, batman.ece.northwestern.edu, robin.ece.northwestern.edu, alfred.ece.northwestern.edu, gordon.ece.northwestern.edu, madhatter.ece.northwestern.edu, joker.ece.northwestern.edu, cobblepott.ece.northwestern.edu, bane.ece.northwestern.edu, nightwing.ece.northwestern.edu, selina.ece.northwestern.edu, ras.ece.northwestern.edu, poisonivy.ece.northwestern.edu, freeze.ece.northwestern.edu, scarecrow.ece.northwestern.edu, clayface.ece.northwestern.edu, harley.ece.northwestern.edu, killercroc.ece.northwestern.edu, huntress.ece.northwestern.edu, batgirl.ece.northwestern.edu, riddler.ece.northwestern.edu, hush.ece.northwestern.edu

• WOT systems
  murphy.wot.ece.northwestern.edu, finagle.wot.ece.northwestern.edu, hanlon.wot.ece.northwestern.edu, moore.wot.ece.northwestern.edu
Outline of today’s CAT

- Structure of the course
- CAT and compilers
- CAT and computer architecture
- CAT and programming language
CS 323 CAT in a nutshell

• About: understanding and transforming code automatically
• Tuesday/Thursday 5pm – 6:20pm

• Umang’s office hours: Friday 1:00pm – 2:00pm in Mudd 3532
• Simone’s office hours: Tuesday 12:00pm – 1:00pm via Zoom

• CAT is on Canvas
  • Materials/Assignments/Grades on Canvas
  • You’ll upload your assignments on Canvas
CAT materials

- Modern compiler implementation
- Slides and assigned papers
- LLVM documentation

http://llvm.org
CAT slides

• You can find last year slides from the class website

• We improve slides every year
  • based on problems we will observe during the next 10 weeks
  • as well as your feedbacks we will ask you at the end
  • Our goal: maximize how much you learn in 10 weeks

• We will upload to Canvas the new version of the slides just before each class

• Slides support my teaching philosophy
The spirit of my lectures a.k.a. my teaching philosophy

• I’ll describe problems/opportunities
• I’ll describe concepts required to solve these problems (take advantage of these opportunities)
• I’ll describe their solutions that are based on these concepts

Problems/opportunities/concepts are structured in weeks

• I’ll describe new problems/opportunities
• You’ll apply concepts/solutions learned during my lectures to solve the new problems/opportunities
  • Required to pass the homework
The CAT structure

Week
1st day
2nd day
Homework

Today
12/8
The CAT grading

• Homework: 100 points
  • 10 points per assignment
  • The first assignment is easy

• Extra points
  • Extra homework
  • Answering (correctly) special questions (I will emphasize them) during lectures
  • Best student so far: **114 points**!
The CAT competition

• At the end, there will be a competition between your CATs

• The team that designed the best CATs
  • Get an A automatically (no matter how many points they have)
  • Their names go to the “hall of fame” of this class
Rules for homework

• You are encouraged (but not required) to work in pairs
  • Pair programming is *not* team programming
  • **Declare your pair by the next lecture (via email to TA)**
    After a pair is formed, you can only split
    (no new pairs will be allowed; also, pairs cannot merge)

• No copying of code is allowed between pairs

• Tool, infrastructure help is allowed
  • First try it on your own
    (google and tool documentation are your friends)

• Avoid plagiarism
  [www.northwestern.edu/provost/policies/academic-integrity/how-to-avoid-plagiarism.html](http://www.northwestern.edu/provost/policies/academic-integrity/how-to-avoid-plagiarism.html)

• If you don’t know, please ask: simone.campanoni@northwestern.edu
Summary

• My duties
  • Teach you code analysis and transformation
  • And how to implement them in a production compiler (LLVM)

• Your duties
  • Learn code analysis and transformation
  • Implement a few of them in LLVM
    • Write code
    • Test your code
    • Then, think much harder about how to actually test your code
    • (Sometimes) Answer my questions about your code

No final exam
Structure & flexibility

• CAT is structured w/ topics

• Best way to learn is to be excited about a topic

• Interested in something?

Speak

I’ll do my best to include your topic on the fly
Week 1

Today
• Welcome/Structure
• Compiler/CAT

Next lecture
LLVM

12/8
Outline of today’s CAT

• Structure of the course

• CAT and compilers

• CAT and computer architecture

• CAT and programming language
The role of compilers

If there is no coffee, if I still have work to do, I’ll keep working, I’ll go to the coffee shop

Will I go to the coffee shop when I have coffee?

Code analysis and transformation
Math

PL

Compilers & CATs

Arch

Practice
Example of CAT

varX = 5

... print varX

...
Example of CAT

```
varX = 5
...
...
...
...
What will it print? print 5  print varX
...
```
Example of CAT

\[ \text{varX} = 5 \]
\[ \ldots \]
\[ \ldots \]
\[ \text{print 5} \]

Is it worth transforming this code?
Designing CATs

• Choose a goal
  • Performance, energy, identifying bugs, discovering code properties, ...

• Design automatic code analyses to obtain the required information

• Occasionally design code transformations
Use of CATs

• Compilers
  • Increase performance
  • Decrease energy consumption
  • Decrease code size
  • Drive the code translation

• Developing tools (e.g., VIM, EMACS)
  • Understanding code (e.g., scopes, variables)

• Computer architecture
Structure of a compiler

Character stream (Source code)

Lexical analysis

Tokens

Syntactic & semantic analysis

AST

```
int main (){
    printf("Hello World\n");
    return 0;
}
```
Structure of a compiler

Character stream (Source code)

Lexical analysis

Tokens

Syntactic & semantic analysis

AST

`int main` ...

`INT SPACE STRING SPACE` ...

Function signature

Return type

INT

Function name

STRING
Structure of a compiler

Syntactic & semantic analysis

AST

IR code generation

IR

Function signature

Return type

INT

Function name

STRING

; Function Attrs: nounwind uwtable
define int @main() {
Structure of a compiler

Character stream (Source code)

Front-end

Middle-end

Back-end

Machine code

CS 322: Compiler Construction

i
n
t
m
a
i
n
...
Structure of a compiler

Character stream (Source code) \rightarrow Front-end \rightarrow IR \rightarrow Middle-end \rightarrow IR \rightarrow Back-end \rightarrow Machine code

Character stream (Source code) \rightarrow Front-end

Middle-end

Back-end

Machine code
Structure of a compiler
Structure of a compiler

C
\downarrow
Front-end
\downarrow IR
Middle-end
\downarrow IR
Back-end
\downarrow Machine code

Java
\downarrow
Front-end
\downarrow
Middle-end
\downarrow
Back-end
\downarrow Machine code
Structure of a compiler

- Front-end
  - IR
  - Middle-end
    - IR
    - Back-end
      - Machine code
  - Machine code
- Java
  - FE
  - M2
  - Machine code
Structure of a compiler

Front-end

Middle-end

Back-end

Machine code

C

IR

Java

IR

FE

BE

M2

Front-end

Middle-end

Back-end

Java

M2
Structure of a compiler

Front-end 1

Middle-end

Front-end 2

Back-end A

Back-end B
Multiple IRs

- Abstract Syntax Tree
- Register-based representation (three-address code)
  \[ R1 = R2 \text{ add } R3 \]
- Stack-based representation
  push 5; push 3; add; pop ;

IR needs to be easy
1) to produce
2) to translate into machine code
3) to transform/optimize
Example of LLVM IR

define i32 @main(i32 %argc, i8** %argv) {
  entry:
    %add = add i32 %argc, 1
    ret i32 %add
}
Multiple IRs used together

Compilation step 1
  IR1
  ↓
Compilation step 2
  IR2
  ↓
Compilation step 3
  ↓
Machine code
Multiple IRs used together

1. Rust
2. Compilation step 1
   - MIR
3. Compilation step 2
   - LLVM IR
4. Compilation step 3
   - Machine code
Multiple IRs used together

L1

Static compiler

IR1

Dynamic compiler FE

IR2

Dynamic compiler BE

Machine code
Multiple IRs used together

Java

Java compiler

Java bytecode

Java VM FE

IR2

Java VM BE

Machine code
CATs that we’ll focus on

• Semantics-preserving transformations
  • Correctness guaranteed

• Goal: performance

• Automatic

• Efficient
Outline of today’s CAT

• Structure of the course

• CAT and compilers

• CAT and computer architecture

• CAT and programming language
Evolution of CATs (hardware point of view)

- Simple hardware (few resources), simple CATs

```
<table>
<thead>
<tr>
<th>Core</th>
<th>Cache L1</th>
<th>Registers</th>
<th>Memory</th>
<th>Cache L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Latency</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

"Moore's Law"
Processor-Memory Performance Gap:
(grows 50% / year)

- CPU: 60%/yr.
- DRAM: 7%/yr.
Evolution of CATs (hardware point of view)

• Simple hardware (few resources), simple CATs
• More hardware resources available to compilers
• Opportunities to improve programs
• Challenging CATs

Compilers/CATs are considered in the processor-design stage!
Evolution of CATs (hardware point of view) (3)

Superscalar

| Inst 1 | Inst 2 | Inst 3 | Inst 4 | Inst 5 | Inst 6 | Inst 7 | Inst 8 |

CATs

Very long instruction word (VLIW)

| Inst 1 | Inst 2 | Inst 3 | Inst 4 | Inst 5 | Inst 6 | Inst 7 | Inst 8 |
Outline of today’s CAT

• Structure of the course

• CAT and compilers

• CAT and computer architecture

• CAT and programming language
Evolution of CATs (PL point of view)

• First electronic computers appeared in the ’40s
• They were programmed in machine language

\[
00101010111001010101001010101011010
\]

• Low level operations only
  • Move data from one location to another
  • Add the contexts of two registers
  • Compare two values

• Programming: slow, tedious, and error prone
Evolution of CATs (PL point of view)

• Low level programming language, simple CATs
  • Not very productive

• More abstraction in programming language, more work for CATs to reduce their performance overhead
  • Macros -> Fortran, Cobol, Lisp -> C, C++, Java, C#, Python, PHP, SQL, ...

• CATs enable new programming languages
Evolution of CATs (PL point of view)

• Abstractions are great for productivity

• CATs remove their overhead

• But abstractions must be carefully evaluated considering CATs

• A simple abstraction in PL can generate challenges for CATs
  • CATs need to be understood
Evolution of CATs (PL point of view) (2)

PL without procedures

void main (){
    Int v1, v2;
    v1 = 1;
    v2 = 2;
    ...
}

Evolution of CATs (PL point of view)(3)

Let’s add procedures to our PL

```c
void myProc (int *a, int *b){...}
myProc(&myVar1, &myVar2);
```
void myProc (int *v1, int *v2) {
    (*v1) = 1;
    (*v2) = 2;
}

What’s the problem for CATs? ... if v1 and v2 alias ...

Understanding if pointers alias: pointer alias analysis

This is one of the most challenging problem in CATs
Conclusion

• CATs used for multiple goals
  • Enable PLs
  • Enable hardware features

• CATs are effected by
  • Their input language
  • The target hardware

• When you design a PL or a new hardware platform, you need to understand what CATs can and can’t do
  • Often: a can’t becomes can thanks to research on CATs
Ideal CATs

• Proved to be correct

• Improve performance of many important programs

• Minor compilation time

• Negligible implementation efforts
Code transformations

• Typically: they preserve the original program semantics
  • These are the transformations that are included in commodity compilers (e.g., gcc, clang, icc)

• In this class, we only consider this type of code transformations
Code transformation

**Code transformation:**
An algorithm that takes code as input and it generates new code as output

![Diagram](Diagram.png)

**Semantically-preserving code transformation:**
A code transformation that *always* generates code that is *guaranteed* to have the *same semantics* of the code given as input.

**What is the program semantics?**
Program semantic

**Program semantic:** Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input.

```c
int main (int argc, char *argv[])
{
    int x = argc;
    int y = x + 1;
    y++;
    printf("%d", x + y);
    return 0;
}
```

```c
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", argc + y);
    return 0;
}
```

```c
int main (int argc, char *argv[])
{
    int y = argc + 2;
    printf("%d", 2*argc + 3);
    return 0;
}
```
Program semantic

Program semantic: Input -> Output
Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input.

```c
int main (  
    int argc, char *argv[]  
){
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 1;
}
```

```c
int main (  
    int argc, char *argv[]  
){
    int y = argc + 2;
    printf("%d", 2*argc + 2);
    return 0;
}
```

```
$ ./myprog 2
6
$ echo $?
```

```
$ ./myprog 2
6
$ echo $?
```
Program semantic

Program semantic: Input -> Output
Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input

```
int main (int argc, char *argv[])
{
    int y = 42;
    return y;
}
```

Our new code transformation

```
int main (int argc, char *argv[])
{
    int y = 42;
    return 42;
}
```

*We have preserved the semantics of the original code!*
Program semantic

**Program semantic:** Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input.

Our transformation needs to understand how the execution flows through the instructions to preserve the semantics!

We haven’t preserved the semantics of the original code

Our new code transformation

```
int main (int argc, char *argv[])
{
    int y = 42;
    int x = y;
    if (argc > 20)
        y = 81;
    return x + y;
}
```

```
int main (int argc, char *argv[])
{
    int y = 42;
    int x = 42;
    if (argc > 20)
        y = 81;
    return x + 42;
}
```

*This is ok!*

*When this is executed*
As Linus Torvalds says ...

Talk is cheap. Show me the code.

Demo time