Optimizing Compilers, Models of Computation

Notes:
• Lab 3 checkoff due Wed.
• Quiz 2 Friday
• Quiz reviews
  • 7:30-9PM
  • Tue: room 4-270
  • Wed: room 4-370

http://xkcd.com/303/
Stack Detective

\( \text{fact}(n) \) is called. During the calculation, the computer is stopped with the PC at 0x40; the stack contents are shown (in hex).

- What’s the argument to the active call to fact? 3
- What’s the argument to the original call to fact? 6
- What’s the location of the original calling (BR) instruction? 80 – 4 = 7C
- What instruction is about to be executed? DEALLOCATE(1)
- What value is in BP? 13C
- What value is in SP? 13C+4+4=144
- What value is in R0? \( \text{fact}(2) = 2 \)
Anatomy of a Modern Compiler

- Read source program
- Break it up into basic elements
- Check correctness, report errors
- Translate to generic intermediate representation (IR)

- Optimize IR
- Translate IR to ASM
- Optimize ASM
Frontend Stages

- Lexical analysis (scanning): Source $\rightarrow$ List of tokens

```java
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

```plaintext
(“int”, KEYWORD)
(“x”, IDENTIFIER)
(“=”, OPERATOR)
(“3”, INT_CONSTANT)
(“,“, SPECIAL_SYMBOL)
(“int”, KEYWORD)
(“y”, IDENTIFIER)
(“=”, OPERATOR)
(“x”, IDENTIFIER)
(“+”, OPERATOR)
(“7”, INT_CONSTANT)
(“,“, SPECIAL_SYMBOL)
(“while”, KEYWORD)
(“(”, SPECIAL_SYMBOL)
...
Frontend Stages

- Lexical analysis (scanning): Source $\rightarrow$ Tokens
- Syntactic analysis (parsing): Tokens $\rightarrow$ Syntax tree

```
Compound statement
  op = var x const 3
  op = var y
  op + var x const 7
  op != var x var y
  while
    test
    body
      if
        test
        then
          op > var x var y
        else
          op = var x var y
        op = var x var y
```

```
Compound statement
  op = var x const 3
  op = var y
  op + var x const 7
  op != var x var y
  while
    test
    body
      if
        test
        then
          op > var x var y
        else
          op = var x var y
        op = var x var y
```
Frontend Stages

- Lexical analysis (scanning): Source \(\rightarrow\) Tokens
- Syntactic analysis (parsing): Tokens \(\rightarrow\) Syntax tree
- Semantic analysis (mainly, type checking)

Consider:

```java
int x = "bananas";
```

Syntax OK
Semantically (meaning) WRONG

**Line 1: error, invalid conversion from string constant to int**
Intermediate Representation (IR)

- Internal compiler language that is:
  - Language-independent
  - Machine-independent
  - Easy to optimize

- Why yet another language?
  - Assembly does not have enough info to optimize it well
  - Enables modularity and reuse
Common IR: Control Flow Graph

- **Assignments:**
  \[ x = a \text{ op } b \]
  - `x`: Variable
  - `a` and `b`: Variable or constant
  - `op`: `+`, `-`, `*`, ...

- **Basic block:** Sequence of assignments with an optional branch at the end
  ```
  x = 3
  y = x + 7
  if (x != y)
  ```

- **Control flow graph:**
  - **Nodes:** Basic blocks
  - **Edges:** branches between basic blocks
Control Flow Graph for GCD

```c
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

Looks like a high-level FSM…
IR Optimization

• Perform a set of passes over the CFG
  – Each pass does a specific, simple task over the CFG
  – By repeating multiple simple passes on the CFG over and over, compilers achieve very complex optimizations

• Example optimizations:
  – Dead code elimination: Eliminate assignments to variables that are never used, or basic blocks that are never reached
  – Constant propagation: Identify variables that are constant, substitute the constant elsewhere
  – Constant folding: Compute and substitute constant expressions
```c
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

**NOTE:** Expressions with > 2 vars or constants broken down in multiple assignments, using temporary variables
Example IR Optimizations

```c
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

1. Dead code elim
2. Constant propagation
3. Constant folding
Example IR Optimizations

```c
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
Example IR Optimizations

```c
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim
Example IR Optimizations

```c
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

Dumb repetition of simple transformations on CFGs

Extremely powerful optimizations

More optimizations by adding passes: Common subexpression elimination, loop-invariant code motion, loop unrolling...

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim
11. Constant propagation
12. Constant folding
13. Dead code elim
14. Constant propagation
15. Constant folding
No changes in 13,14,15 → DONE
Code Generation

- Translate generated IR to assembly

- Register allocation: Map variables to registers
  - If variables > registers, map some to memory, and load/store them when needed

- Translate each assignment to instructions
  - Some assignments may require > 1 instr if our ISA doesn’t have op

- Emit each basic block: label, assignments, and branches

- Lay out basic blocks, removing superfluous jumps

- ISA and CPU-specific optimizations
  - e.g., if possible, reorder instructions to improve performance
Putting It All Together: GCD

Source code

```c
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

IR

```
start
x = 3
y = x + 7
if (x != y)
    T
    if (x > y)
        T
        x = x - y;
    F
    else {
        y = y - x;
    }
    T
    if (x != y)
        T
        end
    F
end
```

Optimized IR

```
start
x = 3
y = 10
if (x > y)
    T
    x = x - y;
    F
    y = y - x;
    T
    if (x != y)
        T
        end
    F
end
```
Putting It All Together: GCD

1. Allocate registers:
   \( x: R_0, y: R_1 \)

2. Produce each basic block:

   **BBL0:**
   - \( \text{CMOVE} (3, R_0) \)
   - \( \text{CMOVE} (10, R_1) \)
   - \( \text{BR} (\text{BBL1}) \)

   **BBL1:**
   - \( \text{CMPLT} (R_1, R_0, R_2) \)
   - \( \text{BT} (R_2, \text{BBL2}) \)
   - \( \text{BR} (\text{BBL3}) \)

   **BBL2:**
   - \( \text{SUB} (R_0, R_1, R_0) \)
   - \( \text{BR} (\text{BBL4}) \)

   **BBL3:**
   - \( \text{SUB} (R_1, R_0, R_1) \)
   - \( \text{BR} (\text{BBL4}) \)

   **BBL4:**
   - \( \text{CMPEQ} (R_1, R_0, R_2) \)
   - \( \text{BT} (R_2, \text{end}) \)
   - \( \text{BR} (\text{BBL1}) \)

3. Lay out BBs, removing superfluous branches:

   **BBL0:**
   - \( \text{CMOVE} (3, R_0) \)
   - \( \text{CMOVE} (10, R_1) \)

   **BBL1:**
   - \( \text{CMPLT} (R_1, R_0, R_2) \)
   - \( \text{BT} (R_2, \text{BBL2}) \)

   **BBL2:**
   - \( \text{SUB} (R_0, R_1, R_0) \)
   - \( \text{BR} (\text{BBL4}) \)

   **BBL3:**
   - \( \text{SUB} (R_1, R_0, R_1) \)
   - \( \text{BR} (\text{BBL4}) \)

   **BBL4:**
   - \( \text{CMPEQ} (R_1, R_0, R_2) \)
   - \( \text{BF} (R_2, \text{BBL1}) \)

   **end:**
Summary: Modern Compilers

**Frontend (analysis)**
- Produces IR if correct program
- Produces meaningful errors

- Source code
- Tokens
- Syntax tree
- Type-checked syntax tree

**Backend (synthesis)**
- Produces optimized program

- High-quality assembly (often > hand-coded!)

---

- Lexical analysis
- Syntactic analysis
- Semantic analysis
- Generate IR
- Optimize IR
- Generate ASM
Universality?

- Recall: We say a set of Boolean gates is universal if we can implement any Boolean function using only gates from that set.

- What problems can we solve with a von Neumann computer? (e.g., the Beta)
  - Everything that FSMs can solve?
  - Every problem?
  - Does it depend on the ISA?

- Needed: a mathematical model of computation
  - Prove what can be computed, what can’t
The roots of computer science stem from the evaluation of many alternative mathematical “models” of computation to determine the classes of computations each could represent.

An elusive goal was to find a universal model, capable of representing all practical computations...

- switches
- gates
- combinational logic
- memories
- FSMs

Are FSMs the ultimate digital computing device?

*We’ve got FSMs... what else do we need?*
FSM Limitations

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM. For instance:

Well-formed Parentheses Checker:
Given any string of coded left & right parens, outputs 1 if it is balanced, else 0.

Simple, easy to describe.

Can this problem be solved using an FSM???

NO!

PROBLEM: Requires arbitrarily many states, depending on input. Must "COUNT" unmatched left parens. An FSM can only keep track of a finite number of unmatched parens: for every FSM, we can find a string it can’t check.

I know how to fix that!

Alan Turing
Turing Machines

Alan Turing was one of a group of researchers studying alternative models of computation.

He proposed a conceptual model consisting of an FSM combined with an infinite digital tape that could be read and written at each step.

- encode input as symbols on tape
- FSM reads tape writes symbols/changes state until it halts
- Answer encoded on tape

Turing’s model (like others of the time) solves the "FINITE" problem of FSMs.

Bounded tape configuration can be expressed as a (large!) integer

We can talk about TM 347 running on input 51, producing an answer of 42.

TMs as integer functions: $y = TM_I[x]$
Other Models of Computation...

Turing Machines [Turing]

Recursive Functions [Kleene]

\[ F(0,x) \equiv x \]
\[ F(1+y,x) \equiv 1+F(x,y) \]

(\text{define} \ (\text{fact} \ n) \ 
\ \ \ (\ldots \ (\text{fact} \ (- \ n \ 1)) \ )

Lambda calculus [Church, Curry, Rosser...]

\[ \lambda x.\lambda y.xx y \]

(\text{lambda}(x)(\text{lambda}(y)(x \ (x \ y))))

Production Systems [Post, Markov]

\[ \alpha \rightarrow \beta \]

IF pulse=0 THEN patient=dead

Emile Post

Alan Turing

Stephen Kleene

Alonzo Church
Computability

FACT: Each model studied is capable of computing exactly the same set of integer functions!

Proof Technique:
Constructions that translate between models

BIG IDEA:
Computability, independent of computation scheme chosen

Church's Thesis:
Every discrete function computable by ANY realizable machine is computable by some Turing machine.

\[ f(x) \text{ computable} \iff \text{for some } k, \text{ all } x \quad f(x) = T_k[x] \]
“special-purpose” Turing Machines....

Multiplication

Sorting

Meanwhile...

Turing machines Galore!

Is there an alternative to infinitely many ad-hoc Turing Machines?
The Universal Function

Here’s an interesting function to explore: the Universal function, $U$, defined by

$$U(k, j) = T_k[j]$$

Could this be computable???

SURPRISE! $U$ is computable by a Turing Machine:

In fact, there are infinitely many such machines. Each is capable of performing any computation that can be performed by any TM!
Universality

What’s going on here?

k encodes a “program” – a description of some arbitrary machine.

j encodes the input data to be used.

\( T_U \) interprets the program, emulating its processing of the data!

**KEY IDEA:** Interpretation.
Manipulate *coded representations* of computing machines, rather than the machines themselves.
Turing Universality

The *Universal Turing Machine* is the paradigm for modern general-purpose computers!

Basic threshold test: Is your computer *Turing Universal*?
- If so, it can emulate every other Turing machine!
- Thus, your computer can compute any computable function

To show your computer is Universal: demonstrate that it can emulate some known UTM.
- Actually given finite memory, can only emulate UTMs + inputs up to a certain size
- This is not a high bar: conditional branches (BEQ) and some simple arithmetic (SUB) are enough.
Coded Algorithms: Key to CS

data vs hardware

Algorithms as data: enables
COMPILERS: analyze, optimize, transform behavior

\[ T_{\text{COMPILER-X-to-Y}}[P_X] = P_Y, \text{ such that } T_X[P_X, z] = T_Y[P_Y, z] \]

SOFTWARE ENGINEERING:
Composition, iteration, abstraction of coded behavior
\( F(x) = g(h(x), p((q(x))) \)

LANGUAGE DESIGN: Separate specification from implementation
- C, Java, JSIM, Linux, ... all run on X86, Sun, ARM, JVM, CLR, ...
- Parallel development paths:
  - Language/Software design
  - Interpreter/Hardware design
Uncomputability (!)

Uncomputable functions: There are well-defined discrete functions that a Turing machine cannot compute

- No algorithm can compute f(x) for arbitrary x in finite number of steps
- Not that we don’t know algorithm - can prove no algorithm exists
- Corollary: Finite memory is not the only limiting factor on whether we can solve a problem

The most famous uncomputable function is the so-called Halting function, \( f_H(k, j) \), defined by:

\[
f_H(k, j) = \begin{cases} 
1 & \text{if } T_k[j] \text{ halts;} \\
0 & \text{otherwise.}
\end{cases}
\]

\( f_H(k, j) \) determines whether the \( k \)th TM halts when given a tape containing \( j \).
Why $f_H$ is Uncomputable

If $f_H$ is computable, it is equivalent to some TM (say, $T_H$):

$k \rightarrow T_H \rightarrow 1$ iff $T_k[j]$ halts, else 0

$j \rightarrow T_H$

Then $T_N$ (N for “Nasty”), which must be computable if $T_H$ is:

$T_N[x]$: LOOPS if $T_x[x]$ halts; HALTS if $T_x[x]$ loops

Finally, consider giving N as an argument to $T_N$:

$T_N[N]$: LOOPS if $T_N[N]$ halts; HALTS if $T_N[N]$ loops

Contradiction! $T_N$ can’t be computable, hence $T_H$ can’t either!