CSC2/455 Software Analysis and Improvement
Intermediate Representations (IRs)

Sreepathi Pai
January 22, 2020

URCS
Outline

Introduction

Midend

Miscellaneous
Introduction

Midend

Miscellaneous
Classic Compiler Architecture

language-dependent frontend

produces PARSE TREE
and ABSTRACT SYNTAX TREE

midend

produces ASSEMBLY

machine-dependent backend
Outline

Introduction

Midend

Miscellaneous
What does the midend do?

- Mostly language and machine independent analyses
- Majority of analyses run in this stage
- Multiple *intermediate representations* used
  - Starts from abstract syntax tree
  - Usually stops before instruction scheduling/register allocation
  - Examples: AST, CFG, DDG, PDG, etc.
- There is no *one* Intermediate Representation
  - although people have tried ...
Running example

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

In Python (assume `math.sqrt` is `sqrt`):

```python
x = (-b - sqrt(b*b - 4*a*c)) / (2*a)
```
Abstract Syntax Tree

```
x / b - \sqrt - 2 a * b * c 4 a
```

```
What does this remind you of?
• Actual ASTs are not binary trees!
  • Will usually have list of descendants instead of left and right
  • descendants may be more specific, while may have condition and body
• ASTs imply treewalking
  • Works best when manipulating source code
  • e.g. Source-to-source compilers
  • Or when control flow is not important
• Other lower level forms are “closer to machine”
  • Stack machines
  • 3 address code
Stack Machine

```
push a
push 2
mul
push c
push a
push 4
mul
mul
push b
push b
mul
sub
sqrt
push b
neg
sub
div
pop x
```

How do you produce stack machine code from an AST?
Generating Stack Machine Code

push a
push 2
mul
push c
push a
push 4
mul
mul
push b
push b
mul
sub
sqrt
push b
neg
sub
div
pop x
Stack machines

- Compact in size
- Easy to execute
- BUT, fixed order of execution
  - Bad for parallelism
- Hard to analyze
- Nevertheless, widely used:
  - Java bytecode
  - WebAssembly
3 address code

- 3 “addresses”
  - Two source operands
  - One destination operand
- Addresses are actually *names* generated by compiler
Our example in 3 address code

Assume machine is not a strict load/store architecture (so memory can be referenced directly):

\[
\begin{align*}
t1 & \leftarrow -b \\
t2 & \leftarrow b \times b \\
t3 & \leftarrow 4 \times a \\
t4 & \leftarrow t3 \times c \\
t5 & \leftarrow t2 - t4 \\
t6 & \leftarrow \sqrt{t5} \\
t7 & \leftarrow t1 - t6 \\
t8 & \leftarrow 2 \times a \\
t9 & \leftarrow t7 / t8 \\
x & \leftarrow t9
\end{align*}
\]

(Here ‘\(-\)’ is \(\leftarrow\), signifying assignment)

How do we produce stack 3 address code from the AST?
Producing 3 address code

\[
\begin{align*}
t_1 & \leftarrow -b \\
t_2 & \leftarrow b \times b \\
t_3 & \leftarrow 4 \times a \\
t_4 & \leftarrow t_3 \times c \\
t_5 & \leftarrow t_2 - t_4 \\
t_6 & \leftarrow \sqrt{t_5} \\
t_7 & \leftarrow t_1 - t_6 \\
t_8 & \leftarrow 2 \times a \\
t_9 & \leftarrow t_7 / t_8 \\
x & \leftarrow t_9
\end{align*}
\]
Data Dependence Graphs (DDGs)

- DDGs track “data flow” as an acyclic graph
- Strict (partial) order in which operations must be performed
  - Can’t use a value that has not been calculated yet!
- But multiple orders may be allowed!
  - Topological sort
- Will revisit DDGs when we discuss instruction scheduling
\begin{align*}
t1 & \leftarrow -b \\
t2 & \leftarrow b \times b \\
t3 & \leftarrow 4 \times a \\
t4 & \leftarrow t3 \times c \\
t5 & \leftarrow t2 - t4 \\
t6 & \leftarrow \sqrt{t5} \\
t7 & \leftarrow t1 - t6 \\
t8 & \leftarrow 2 \times a \\
t9 & \leftarrow t7 / t8 \\
x & \leftarrow t9
\end{align*}
What should the 3 address code for the code below look like?

```python
if n % 2 == 0:
    n = n / 2
else:
    n = 3 * n + 1
```
• An if AST node has a condition, true-code, and false-code
3 Address Code for if

\[ t_1 \leftarrow n \mod 2 \]
\[ \text{if (} t_1 \neq 0 \text{) goto L1} \]

\[ t_2 \leftarrow n / 2 \]
\[ n \leftarrow t_2 \]
\[ \text{goto L2} \]

L1:
\[ t_3 \leftarrow 3 \times n \]
\[ t_4 \leftarrow t_3 + 1 \]
\[ n \leftarrow t_4 \]

L2:

- 3 address code can contain:
  - (simplified) conditional branches
  - unconditional branches
Control Flow Graphs (CFGs)

- “Hybrid” representation
  - Linear code + Graph structure
- Each node in the CFG is a “basic block”
  - Linear code
  - Single entry, single exit
  - “Straight-line code”
- Most common form for analysis

ENTRY
  t1 <- n % 2
  t1 == 0
EXIT
  t2 <- n / 2
  n <- t2
  t3 <- 3 * n
  t4 <- t3 + 1
  n <- t4
In this course

• We will write midend + some bits of a backend
• Input language: C
• Output language: C in 3 address form
  • Not assembly (maybe extra credit?)
• Using Python library pycparser
For assignments, make sure to review

- Basic data structures
  - lists
  - trees
  - graphs
- Basic data structure traversals
  - Infix, prefix, postfix
  - Depth-first, breadth-first
- And how to implement them in Python
  - Using Python standard libraries is fine
Outline

Introduction

Midend

Miscellaneous
What is this code from TensorFlow doing?

```python
a = tf.constant(2)
b = tf.constant(3)

with tf.Session() as sess:
    print("a=2, b=3")
    print("Addition with constants: %i" % sess.run(a+b))
    print("Multiplication with constants: %i" % sess.run(a*b))
```

https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/1_Introduction/basic_operations.py
Metaprogramming

- TensorFlow API builds a graph
  - directed, acyclic
  - similar to the DDG
  - very common technique
- When `sess.run` is called, graph is compiled and executed
- Advantages:
  - No syntax, no parsing!
- Disadvantages:
  - ?
• Chapter 5 of Cooper and Turczon
  • Up to 5.4 in this lecture, but we will ultimately study the whole chapter

• (Optional) For a different perspective, Chapter 4 of Muchnick
  • Chapters 1–3 are essentially a recap of CSC 2/454