CMSC 631 – Program Analysis and Understanding
Spring 2013
What you’ll learn

- Formal systems and notations
  - Vocabulary for talking about programs

- Program analysis
  - Automatic reasoning about source code

- Programming language features
  - Affects programs and how we reason about them
How you’ll learn it

• Implement
  ▪ You will build some program analyzers as projects using the Objective Caml programming language

• Prove
  ▪ You will employ mathematics to reason about programs
    - Most work will be “pen and paper”
    - We will also introduce you to interactive theorem proving: technology ensures your proofs are actually correct!

• Take it one step further: substantial final project
What you’ll gain

• Better programming ability
  ▪ By understanding programming languages deeply

• Mathematical methods and maturity
  ▪ Formalization and proof techniques will transfer to other areas

• How to program in OCaml and use an SMT solver
  ▪ Both skills are increasingly useful/important
Personnel

• **Michael Hicks** (Assoc. Prof)
  - 4131 AVW, mwh@cs.umd.edu

• **Jeff Foster** (Assoc. Prof)
  - 4129 AVW, jfoster@cs.umd.edu

• **Matthew Hammer** (Post-doc)
  - 4161 AVW, hammer@cs.umd.edu

• **Stevie Strickland** (Post-doc)
  - 4161 AVW, sstrickl@cs.umd.edu

• **TA: Piotr Mardziel**, piotrm@cs.umd.edu
Prerequisite
Prerequisite

• CMSC 430 or equivalent
Prerequisite

- CMSC 430 or equivalent
Prerequisite

• CMSC 430 or equivalent
  ▪ Ideas we will use in this class:
    - Parse trees/abstract syntax trees
    - BNF notation for grammars
    - Programming language maturity
      - Familiarity with several different languages/paradigms
    - General information about programming language design
  ▪ Talk to me if you’re not sure
Textbooks

• No required textbooks
  ▪ But see web page for suggestions
  ▪ Recommended text:
    - Pierce, *Types and Programming Languages*
  ▪ A second book, also good:
    - Huth and Ryan, *Logic in Computer Science*

• Neither covers everything in the course
• Recommended two on reserve in CS library
  ▪ Though CS library’s days are limited
Forum

• Piazza
  ▪ https://piazza.com/umd/spring2013/cmsc631
  ▪ Need to sign up

• Can use piazza to ask and answer questions about lectures, assignments, etc.
  ▪ Please use this forum unless you have personal request (e.g., about your grade, an absence, etc.)
Expectations: Homework (45%)

• Programming assignments
  ▪ OCaml programming warmup
  ▪ Implementing a symbolic executor, type inference engine, and one other (TBD)

• Written assignments
  ▪ From basic mathematics to
  ▪ methods for expressing a program’s semantics to
  ▪ methods for proving properties about programs
Late Policy on Assignments

• Programming assignments: Due at midnight
  ▪ Submit via the submit server (see class web page)

• No late submissions
  ▪ Contact instructors as soon as possible about extenuating circumstances
    - E.g., religious holidays
Expectations: Participation (5%)

• Will need to read some papers for class
  ▪ Scattered through the semester
  ▪ Should come prepared to contribute to discussion
Expectations: Project (25%)

• Class goal: Teach you how to do research
  - So you have to do research as part of the class

• Substantial research project (25% of grade)
  - Any topic vaguely related to the class is acceptable
    - Will post some suggestions for projects later on
    - May also be able to share project with other class
  - Completed in groups of size 2 (possibly 1 or 3)

• Will occupy the latter 2/3 of semester
  - But will still have some Coq assignments
Expectations: Project (cont’d)

• Deliverables
  - Project proposal (one page) + talk with instructors
  - Project write-up
    - A conference-style paper (5-15 pages, as appropriate)
  - Implementation, if any
  - In-class presentation (last 3 days of class)
    - 15-20 minutes, depending on # of projects

• In the past, several 631 projects led to papers
  - Not required (!), but possible
Expectations: Exam (25%)

• Final exam
  ▪ Based on course assignments
  ▪ Take home exam
    - The exam will be available for 96 hours
    - You pick a 48-hour window during that time during which to take the exam
  ▪ Dates will fall during the final exam period
Academic Dishonesty

• Don’t do it

• See the syllabus on the web page for
  ▪ a careful description of what constitutes academic dishonesty
  ▪ the consequences of being caught
A whirlwind tour
Abstract Interpretation

• Rice’s Theorem: Any non-trivial property of programs is undecidable
  ▪ Uh-oh! We can’t do anything. So much for this course...

• Need to make some kind of approximation
  ▪ Abstract the behavior of the program
  ▪ ...and then analyze the abstraction

• Seminal papers: Cousot and Cousot, 1977, 1979
Example

\( e ::= n \mid e + e \)

\[ \alpha(n) = \begin{cases} 
- & n < 0 \\
0 & n = 0 \\
+ & n > 0 
\end{cases} \]

\[ \begin{array}{cc|ccc}
+ & - & 0 & + & ? \\
- & - & - & - \\
0 & - & 0 & + \\
+ & ? & + & + 
\end{array} \]

- Notice the need for ? value
- Arises because of the abstraction
Dataflow Analysis

- Classic style of program analysis
- Used in optimizing compilers
  - Constant propagation
  - Common sub-expression elimination
  - Loop unrolling and code motion
  - etc.
- Efficiently implementable
  - At least, *intraprocedurally* (within a single proc.)
  - Use bit-vectors, fixpoint computation
Control-Flow Graph

\[ x := 3 \]
\[ \text{if} \ (x) \ \text{then} \]
\[ y := z + w \]
\[ \text{else} \]
\[ L: \{ y := 0 \} \]
\[ x := 2 * x \]
\[ \text{if} \ (x) \ \text{then goto L} \]
\[ x := 3 \]
if (!x) then
  \[ y := z + w \]
else
  L: \{ y := 0 \}
\[ x := 2 \times x \]
if (!x) then goto L
Control-Flow Graph

x := 3

y := z + w

y := 0

x := 2 * x
Control-Flow Graph

- $x := 3$
- $y := z + w$
- $y := 0$
- $x := 2 \times x$
- $x = *$
Control-Flow Graph

x := 3

y := z + w

y := 0

x := 2 * x

x = *

x = 3
Control-Flow Graph

x := 3

y := z + w

y := 0

x := 2 * x

x = *

x = 3
Control-Flow Graph

\[
\begin{align*}
x &:= 3 \\
y &:= z + w \\
x &:= 2 \times x \\
y &:= 0 \\
x &:= \ast
\end{align*}
\]
Control-Flow Graph

x := 3

y := z + w

y := 0

x := 2 * x

x = *

x = 3

x = 3

x = 3

x = 3
Control-Flow Graph

\[ x := 3 \]
\[ y := z + w \]
\[ y := 0 \]
\[ x := 2 \times x \]
\[ x = * \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = 6 \]
Control-Flow Graph

x := 3

y := z + w

y := 0

x := 2 * x

x = *

x = 3

x = 3

x = 3

x = 6
Control-Flow Graph

\[ x := 3 \]
\[ y := z + w \]
\[ y := 0 \]
\[ x := 2 \times x \]
\[ x = 6 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = 3 \]
\[ x = ? \]
\[ x = * \]
Control-Flow Graph

\[
\begin{align*}
&x := 3 \\
&y := z + w \\
&y := 0 \\
&x := 2 \times x \\
&x = 3 \\
&x = 3 \\
&x = ? \\
&x = * \\
&x = ? \\
&x = 6 \\
&x = ? \\
&x = 3 \\
&x = 6
\end{align*}
\]
Control-Flow Graph

- $x := 3$
- $y := z + w$
- $y := 0$
- $x := 2 \times x$
- $x = *$
- $x = ?$
- $x = 3$
- $x = ?$
- $x = 6$
Control-Flow Graph

\begin{align*}
x & := 3 \\
y & := z + w \\
x & = 3 \\
x & := 2 \times x \\
x & := 0 \\
x & = ? \\
x & = ? \\
x & = *
\end{align*}
Control-Flow Graph

\begin{center}
\begin{tikzpicture}[auto, node distance=2cm, on grid]
    \node[rectangle, draw] (x) {$x := 3$};
    \node[rectangle, draw, below of=x, xshift=-2cm] (y) {$y := z + w$};
    \node[rectangle, draw, below of=x, xshift=2cm] (z) {$y := 0$};
    \node[rectangle, draw, below of=y] (a) {$x := 2 \times x$};
    \node[rectangle, draw, below of=z] (b) {};\node (bl) at (b){};
    \path[->] (x) edge node {$x = 3$} (y);
    \path[->] (x) edge node {$x = *$} (z);
    \path[->] (y) edge node {$x = 3$} (a);
    \path[->] (z) edge node {$x = 0$} (a);
    \path[->] (a) edge node {$x = ?$} (x);
    \path[->] (a) edge [loop below] node {$x = ?$} (a);
\end{tikzpicture}
\end{center}
Control-Flow Graph

\[
\begin{align*}
  x &:= 3, \\
  y &:= z + w, \\
  y &:= 0, \\
  x &:= 2 \times x, \\
  x &= *, \\
  x &= 3, \\
  x &= ?, \\
  x &= ?, \\
  x &= ?, \\
  x &= ?.
\end{align*}
\]
Lattices and Termination

• Dataflow facts form a lattice

\[ \begin{align*}
& x = ? \\
& x = 3 \quad x = 6 \quad \ldots \\
& x = * 
\end{align*} \]

• Each statement has a transformation function

  - \[ \text{Out}(S) = \text{Gen}(S) \cup (\text{In}(S) - \text{Kill}(S)) \]

• Terminates because
  - Finite height lattice
  - Monotone transformation functions
Static Single Assignment Form

- Transform CFG so each use has a single defn

\[
x := 0
\]
\[
v := 3
\]
\[
v := 4 + x
\]
\[
x := x + v
\]

\[
x_1 := 0
\]
\[
v_1 := 3
\]
\[
v_2 := 4 + x_1
\]
\[
v_3 := \Phi(v_1, v_2)
\]
\[
x_2 := x_1 + v_3
\]
Symbolic Execution

• Testing works
  - But, each test only explores one possible execution
    - assert(f(3) == 5)
  - We hope test cases generalize, but no guarantees

• Symbolic execution generalizes testing
  - Allows unknown symbolic variables in evaluation
    - y = α; assert(f(y) == 2*y-1);
  - If execution path depends on unknown, conceptually fork symbolic executor
    - int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }
Symbolic Execution Example

```c
1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10. }
11. assert(x+y+z!=3)
```
1. `int a = α, b = β, c = γ;`
2. `// symbolic`
3. `int x = 0, y = 0, z = 0;`
4. `if (a) {
   5.    x = -2;
   6. }
7. if (b < 5) {
   8.    if (!a && c)  { y = 1; }
   9.    z = 2;
10. }
11. assert(x+y+z!=3)`

x=0, y=0, z=0
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }  
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10.}
11. assert(x+y+z!=3)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10. }
11. assert(x+y+z!=3)

x=0, y=0, z=0
\[ x = -2 \]
Symbolic Execution Example

1. int a = \( \alpha \), b = \( \beta \), c = \( \gamma \);
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10.}
11. assert(x+y+z!=3)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.    x = -2;
6. }
7. if (b < 5) {
8.    if (!a && c) { y = 1; }
9.    z = 2;
10. }
11. assert(x+y+z!=3)
Symbolic Execution Example

1. `int a = α, b = β, c = γ;`
2. // symbolic
3. `int x = 0, y = 0, z = 0;`
4. if (a) {
5.   `x = -2;`
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   `z = 2;`
10. }
11. assert(x+y+z!=3)
1. int a = α, b = β, c = γ;
2.           // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10.}
11. assert(x+y+z!=3)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.    x = -2;
6. }
7. if (b < 5) {
8.    if (!a && c) { y = 1; }
9.    z = 2;
10.}
11. assert(x+y+z!=3)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10. }
11. assert(x+y+z!=3)

\( x=0, y=0, z=0 \)

\( x=-2 \)

\( \beta<5 \)

\( z=2 \)✔

\( \alpha \land (\beta \geq 5) \)✔

\( \alpha \land (\beta<5) \)

path condition
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10. }
11. assert(x+y+z!=3)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10.}
11. assert(x+y+z!=3)
Symbolic Execution Example

1. `int a = α, b = β, c = γ;`
2. `// symbolic`
3. `int x = 0, y = 0, z = 0;`
4. `if (a) {`
5. `  x = -2;`
6. `}
7. `if (b < 5) {
8. `  if (!a && c) { y = 1; }
9. `  z = 2;
10. `}
11. `assert(x+y+z!=3)`

path condition

x=0, y=0, z=0

¬α∧(β≥5)

α∧(β<5)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10. }
11. assert(x+y+z!=3)
Symbolic Execution Example

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10. }
11. assert(x+y+z!=3)
Lambda Calculus

• Three syntactic forms

\[ e ::= x \quad \text{variable} \]

\[ \lambda x.e \quad \text{function} \]

\[ e_1 e_2 \quad \text{function application} \]

• One reduction rule

\[ (\lambda x.e_1) e_2 \rightarrow e_1[e_2\rightarrow x] \quad \text{(replace } x \text{ by } e_2 \text{ in } e_1) \]

• Can represent any computable function!
Example

• Conditionals
  - true = \( \lambda x.\lambda y.x \)  false = \( \lambda x.\lambda y.y \)
  - if a then b else c = \( a \ b \ c \)
    - if true then b else c = \( (\lambda x.\lambda y.x) \ b \ c \rightarrow (\lambda y.b) \ c \rightarrow b \)
    - if false then b else c = \( (\lambda x.\lambda y.y) \ b \ c \rightarrow (\lambda y.y) \ c \rightarrow c \)

• Can also represent numbers, pairs, data structures, etc, etc.

• Result: Lingua franca of PL
ML: Meta-Language

- ML designed originally for theorem provers
  - But after a while, realized could be general-purpose

- Mostly-functional language
  - Similar to lambda-calculus
    - Mostly functional, encouraged not to use side-effects
    - Call-by-value

- We’ll use OCaml for programming assignments
Program Semantics

• To analyze programs, we must know what they mean
  ▪ *Semantics* comes from the Greek *semaino*, “to mean”

• Most language semantics *informal*. But we can do better by making them *formal*. Three styles:
  ▪ Operational semantics (major focus)
    - Like an interpreter
  ▪ Denotational semantics
    - Like a compiler
  ▪ Axiomatic semantics
    - Based on what you can prove about programs
Operational Semantics

• Evaluation is described as transitions in some abstract machine
  ▪ Example: Beta reduction rule from lambda calculus
    \[ (\lambda x. e_1) e_2 \rightarrow e_1[e_2\mapsto x] \]
  ▪ State of machine described by current expression

• There are different styles of abstract machines
  ▪ Small-step (as above), big-step, etc

• The meaning of a program is its fully reduced form (a.k.a. a value)

• This is the most popular style of semantics
Denotational Semantics

• The meaning of a program is defined as a mathematical object, e.g., a function or number

• Typically define an interpretation function \[ \[ \]
  ▪ Program fragment as argument and returns meaning
  ▪ E.g., \[ 3+4 \] = 7

• Gets interesting when we try to find denotations of loops or recursive functions
Denotational Semantics Example

- $b ::= \text{true} \mid \text{false} \mid b \lor b \mid b \land b$
- $e ::= 0 \mid 1 \mid \ldots \mid e + e \mid e \ast e$
- $s ::= e \mid \text{if } b \text{ then } s \text{ else } s$
- Semantics:
  - $\llbracket \text{true} \rrbracket = \text{true}$
  - $\llbracket b_1 \lor b_2 \rrbracket = \begin{cases} \text{true} & \text{if } \llbracket b_1 \rrbracket = \text{true} \text{ or } \llbracket b_2 \rrbracket = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$
  - $\llbracket \text{if } b \text{ then } s_1 \text{ else } s_2 \rrbracket = \begin{cases} \llbracket s_1 \rrbracket & \text{if } \llbracket b \rrbracket = \text{true} \\ \llbracket s_2 \rrbracket & \text{if } \llbracket b \rrbracket = \text{false} \end{cases}$
Axiomatic Semantics

• Operational and denotational semantics let us reason about the meaning of a program
  ▪ Are two programs equivalent? Does a program terminate? Does a program implement a particular specification

• Axiomatic semantics define a program’s meaning in terms of what one can prove about it
  ▪ Due to Floyd, Hoare, Dijkstra, Gries, others
Hoare Triples

- \( \{P\} \ S \ {Q}\)
  - If statement \( S \) is executed in a state satisfying precondition \( P \), then \( S \) will terminate, and \( Q \) will hold of the resulting state
  - Partial correctness: ignore termination

- Semantics defined as proof rules
Proofs of Hoare Triples

• Example rules
  ▪ Assignment: \( \{Q[E \mapsto x]\} \ x := E \ {Q} \)
  ▪ Conditional: \( \{P \land B\} \ S1 \ {Q} \quad \{P \land \neg B\} \ S2 \ {Q} \)
    \[ \{P\} \text{ if } B \text{ then } S1 \text{ else } S2 \ {Q} \]

• Example proof (simplified)

\[
\begin{align*}
\{y>3\} \ x := y \ {x>3} & \quad \{\neg(y>3)\} \ x := 4 \ {x>3} \\
\{\} \text{ if } y>3 \text{ then } x := y \text{ else } x := 4 \ {x>3}
\end{align*}
\]
Type Systems

• A type system is
  ▪ a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. --Pierce

• They are good for
  ▪ Detecting errors (don’t add an integer and a string)
  ▪ Abstraction (hiding representation details)
  ▪ Documentation (tersely summarize an API)

• Designs trade off efficiency, readability, power
Simply-typed $\lambda$-calculus

$e ::= x \mid n \mid \lambda x:\tau.e \mid e\ e$

$\tau ::= \text{int} \mid \tau \to \tau$

$A \vdash e : \tau$ in type environment $A$, expression $e$ has type $\tau$

$\frac{}{A \vdash n : \text{int}}$

$\frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$

$\frac{}{A[\tau\backslash x] \vdash e : \tau'}$

$\frac{}{A \vdash \lambda x:\tau.e : \tau \to \tau'}$

$\frac{A \vdash e1 : \tau \to \tau' \quad A \vdash e2 : \tau}{A \vdash e1\ e2 : \tau'}$
Subtyping

• Liskov:
  - If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( o_1 \), the behavior of \( P \) is unchanged when \( o_2 \) is substituted for \( o_1 \) then \( S \) is a subtype of \( T \).

• Informal statement
  - If anyone expecting a \( T \) can be given an \( S \) instead, then \( S \) is a subtype of \( T \).
Other Technologies and Topics

• Dependent type systems
• Contract systems, gradual typing
• Constraint-based analysis
• Alias and pointer analysis
• Probabilistic computation
• Software synthesis
• Incremental computation
Applications: Dataflow analysis

• Optimizing compilers
  - i.e., any good compiler (gcc, LLVM)

• ESP: Path-sensitive program checker (Microsoft)
  - Example: can check for correct file I/O properties, like files are opened for reading before being read

• FindBugs (UMD): applies dataflow analysis to discover null pointer and other bugs in JVM programs
Applications: Abstract Interpretation

• Terminator (Microsoft)
  - Analyzes code to prove that it terminates (!)
  - Applied to device drivers for Windows kernel
    - Tricky part is reasoning about the heap

• ASTREE (INRIA and others)
  - Used to detect all possible runtime failures (divide by zero, null pointer deref, array out of bounds) on embedded code
  - Used regularly on Airbus avionics software
Applications: Symbolic Execution

• SAGE (Microsoft)
  - Used as a fuzz tester to find buffer overruns etc. in file parsers

• KLEE (Imperial), Mayhem (CMU) Otter, Rubyx, ...
  - Research systems used to enforce security specifications, find vulnerabilities, explore configuration spaces, and more
Applications: Axiomatic Semantics

• Dafny (Microsoft)
  - Can perform deep reasoning about programs
  - Array out-of-bounds
  - Null pointer errors
  - Failure to satisfy internal invariants
  - Employs the Z3 SMT solver

• Long line of other tools, e.g., Spec#, ESC/Java, and more
Applications: Type Systems

- Type qualifiers (in C and Java)
  - Format-string vulnerabilities, deadlocks, file I/O protocol errors, kernel security holes

- Jif (Java+Information Flow)
  - Annotate standard types with additional security labels, where type correctness implies correct protection of sensitive data

- Dependent type systems (e.g., in Coq, Fine, or Agda) can be used to prove deep(er) program properties
Conclusion

• PL has a great mix of theory and practice
  ▪ Very deep theory
  ▪ But lots of practical applications

• Recent exciting new developments
  ▪ Focus on program correctness (and security)
    - instead of speed
  ▪ Scalability to large programs
  ▪ In greater use in mainstream development