Contact Information

- Office: 2112 Siebel Center
- Office hours:
  - Wednesday, Friday 12:50pm – 1:45pm
  - Also by appointment
- Email: egunter@illinois.edu

Course Website

- **main** page - summary of news items
- **policy** - rules governing the course
- **lectures** - syllabus, slides and example code
- **mps** - Information about homework
- **unit projects** - for 4 credit students
- **resources** - papers, tools, and helpful info
- **faq** - answers to some general questions about the course and course resources

Some Course References

- No Required Textbook
  - *Concrete Semantics With Isabelle/HOL*, by Tobias Nipkow and Gerwin Klein. Springer, 2014. (In your svn directory)
  - Lecture Notes of Grigore Rosu, found in Resources

Main Programming Platform: Isabelle/HOL

- Download from: [http://www.cl.cam.ac.uk/research/hvg/Isabelle/](http://www.cl.cam.ac.uk/research/hvg/Isabelle/)
- Runs inside jEdit
- Two implementation languages: SML (for proofs) and Scala (for jEdit)
- Full-powered general-purpose interactive mathematical theorem prover
- Can export executable specifications directly to code in SML, OCaml, Haskell, and Scala.
- Can export annotated theories to \LaTeX documents
- Accompanied by an impressive library of formally verified mathematics and support for programming language semantics in the Archive of Formal Proofs [https://www.isa-afp.org](https://www.isa-afp.org)

Course Grading

- Homeworks – 30%
  - MPs turned in as plain text (theory) files
  - submitted via course svn student directories
- Midterm – 30%
- Final – 40%
- Unit Project
  - Only for 4-credit graduate students
  - Worth 25%, with all other parts scaled down accordingly
Collaboration on Assignments

- You may discuss homeworks and their solutions with others.
- You may work in groups, but you must list members with whom you worked.
- Each student must turn in their own solution separately.
- You may look at examples from class and other similar examples from any source.
- Note: University policy on plagiarism still holds.
- Problems from homework may appear verbatim, or with some modification on exams.

Default Unit Project

1. Design, formalize and create an interpreter for a new language with specified features.
   - Will be an extension of previously describe language.
2. Give a substantial proof in Isabelle of a property of a programming language.
   - Based on language described in class (unless you want to do more).
3. Students may develop alternate projects with instructor approval.

Course Objectives

- Learn different methods of specifying the meaning of language features and how to reason about them.
  - Structural Operational Semantics
  - Transition Semantics
  - Reduction Semantics with Evaluation Contexts
  - Brief overview of K
- Learn to specify different language features.
  - Imperative Features
  - Object Oriented Features
  - Functional Features
  - Type Systems

Semantics

- Expresses the meaning of syntax.
- Static semantics
  - Meaning based only on the form of the expression without executing it.
  - Usually restricted to type checking / type inference.
- Dynamic semantics
  - Method of describing meaning of executing a program.
  - Used for formal reasoning about programs and languages.
  - Several different types:
    - Operational Semantics
    - Axiomatic Semantics
    - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics.
- Different types of semantics serve different purposes.

Operational Semantics

- Start with a simple notion of machine.
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (i.e., following the structure of the program).
- Meaning of program is how its execution changes the state of the machine.
- Useful as basis for implementations.
Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from axioms and inference rules
- Mainly suited to simple imperative programming languages

Axiomatic Semantics is used to formally prove a property (post-condition) of the state (the values of the program variables) after the execution of program, assuming another property (precondition) of the state before execution.

Written:

\[ \{ \text{Precondition} \} \text{Program} \{ \text{Postcondition} \} \]

Source of idea of loop invariant

Useful for program specification and verification

Denotational Semantics

- Construct a function \( M \) assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Mainly used for proving properties of programs

Natural Semantics

- Aka “Big Step Semantics”
- Originally introduced by Giles Kahn
- Provide value for a program by rules and derivations
- Rule conclusions look like:
  \[ (C, m) \Downarrow m' \]
  or
  \[ (E, m) \Downarrow v \]
- Type derivation rules often take very similar shape

Simple Imperative Programming Language #1

\[
\begin{align*}
I & \in \text{Identifiers} \\
N & \in \text{Numerals} \\
E & ::= N \mid I \mid E + E \mid E \times E \mid E - E \\
B & ::= \text{true} \mid \text{false} \mid B \& B \mid B \text{ or } B \mid \text{not } B \\
& \quad \mid E < E \mid E = E \\
C & ::= \text{skip} \mid C \mid \{C\} \mid I ::= E \\
& \quad \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \\
& \quad \mid \text{while } B \text{ do } C \text{ od}
\end{align*}
\]

Natural Semantics of Atomic Expressions

Let \( m : \text{Identifiers} \rightarrow \text{Values} \) be a partial function supplying values for program variable names

- Identifiers: \( (I, m) \Downarrow m(I) \)
- Numerals are values: \( (N, m) \Downarrow N \)
- Booleans: \( \text{true}, \text{false} \Downarrow \text{true} \), \( \text{false} \Downarrow \text{false} \)
### Boolean Expressions

\[
\begin{align*}
(B, m) \downarrow & \text{false} & (B', m) \downarrow & b \\
(B \& B', m) \downarrow & \text{false} & (B \& B', m) \downarrow & b \\
(B, m) \downarrow & \text{true} & (B', m) \downarrow & b \\
(B \lor B', m) \downarrow & \text{true} & (B, m) \downarrow & \text{false} \\
(B, m) \downarrow & \text{false} & (B', m) \downarrow & \text{true}
\end{align*}
\]

- By \( U \sim V = b \), we mean does (the meaning of) the relation \( \sim \) hold on the meaning of \( U \) and \( V \).
- May be specified by a mathematical expression/equation or rules matching \( U \) and \( V \).

### Arithmetic Expressions

\[
(E, m) \downarrow U \quad (E', m) \downarrow V \\
U \oplus V = N
\]

\[
(E \oplus E', m) \downarrow N
\]

where \( N \) is the specified value for \( U \oplus V \).

### Commands

- **Skip:** \( (\text{skip}, m) \downarrow m \)
- **Assignment:** \( \{ I ::= E, m \} \downarrow m[I \leftarrow V] \)
- **Sequencing:** \( (C, m) \downarrow m' \quad (C', m') \downarrow m'' \) \( \{C, C', m \} \downarrow m'' \)
- **Block:** \( m[l \leftarrow V](J) = \begin{cases} V & \text{if } J = l \\ m(J) & \text{otherwise} \end{cases} \)

### If Then Else Command

\[
\begin{align*}
(B, m) \downarrow & \text{true} & (C, m) \downarrow m' \\
\text{if } B \text{ then } C \text{ else } C', m \downarrow m'
\end{align*}
\]

\[
\begin{align*}
(B, m) \downarrow & \text{false} & (C', m) \downarrow m' \\
\text{if } B \text{ then } C' \text{ else } C, m \downarrow m'
\end{align*}
\]

### While Command

\[
\begin{align*}
(B, m) \downarrow & \text{false} & (\text{while } B \text{ do } C \text{ od }, m) \downarrow m \\
(B, m) \downarrow & \text{true} & (C, m) \downarrow m' \quad (\text{while } B \text{ do } C \text{ od }, m') \downarrow m'' \\
\text{if } B \text{ then } C \text{ else } C', m \downarrow m'
\end{align*}
\]

\[
\begin{align*}
\text{while } B \text{ do } C \text{ od }, m \downarrow m''
\end{align*}
\]

### Relations

\[
(E, m) \downarrow U \quad (E', m) \downarrow V \quad U \sim V = b
\]

\[
(E \sim E', m) \downarrow b
\]
Simple Imperative Programming Language #2

I ∈ Identifiers
N ∈ Numerals
E ::= N | I | E + E | E * E | E − E | I ::= E
B ::= true | false | B & B | B or B | not B
    | E < E | E = E
C ::= skip | C; C | {C} | E
    | if B then C else C fi
    | while B do C od

- What is the new semantics?
- What changes to our “machine state” do we need to make?