CIS 500

Software Foundations

Fall 2004

8 September

Course Overview

What is “software foundations”?

Software foundations (a.k.a. “theory of programming languages”) is the study of the meaning of programs.

The goal is finding ways to describe program behaviors that are both precise and abstract.

- Precise because we would like to prove things about how programs behave.
- Abstract because we would like the techniques that we use to apply to lots of different programs, and lots of different programming languages.

Why study software foundations?

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)

  Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)

  Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive

- To develop intuitions for informal reasoning about programs

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)

  Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive

- To develop intuitions for informal reasoning about programs

- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)

  Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive

- To develop intuitions for informal reasoning about programs

- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)

- To understand language features (and their interactions) deeply and develop principles for better language design

Why study software foundations?

- To be able to prove specific facts about particular programs (i.e., program verification)

  Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- To develop intuitions for informal reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- To understand language features (and their interactions) deeply and develop principles for better language design

  PL is the "materials science" of computer science...

What you can expect to get out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - How to view programs and whole languages as formal, mathematical objects
  - How to make and prove rigorous claims about them
  - Detailed study of a range of basic language features
- Deep intuitions about key language properties such as type safety
- Powerful tools for language design, description, and analysis

N.b.: most good software designers are language designers!

What this course is not

- An introduction to programming (if this is what you want, you should be in CIT 591)
- A course on functional programming (though we'll be doing some functional programming along the way)
- A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)
- A comparative survey of many different programming languages and styles (boring!)
- A seminar on programming language research (see CIS 670, MW 1:30-3:00, Moore 212!)

Approaches

"Program meaning" can be approached in many different ways.

Approaches

"Program meaning" can be approached in many different ways.

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.

Approaches

"Program meaning" can be approached in many different ways.

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on systems of logical rules for reasoning about programs.

Approaches

"Program meaning" can be approached in many different ways.

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on systems of logical rules for reasoning about programs.
- Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.

Approaches

"Program meaning" can be approached in many different ways.

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on systems of logical rules for reasoning about programs.
- Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.
- Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.

Approaches

"Program meaning" can be approached in many different ways.

- Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- Program logics such as Hoare logic and dependent type theories focus on systems of logical rules for reasoning about programs.
- Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.
- Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.
- Type systems describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview

In this course, we will concentrate on operational techniques and type systems.

- Part O: Background
  - A taste of OCaml
  - Functional programming style
- Part I: Modelling programming languages
  - Syntax and operational semantics
  - Inductive proof techniques
  - The lambda-calculus
  - Syntactic sugar; fully abstract translations

- Part II: Type systems
  - Simple types
  - Type safety
  - References
  - Subtyping
- Part III: Object-oriented features (case study)
  - A simple imperative object model
  - An analysis of core Java

Administrative Stuff

Personnel

Send email all staff: `cis500@cis.upenn.edu`

Instructor: Stephanie Weirich
Levine 510
`sweirich@cis.upenn.edu`
Office hours today:
Wed, 3:00–4:00
No office hours next week
Office hours, beginning in two weeks:
Wed, 5:00–6:00 and Thu, 4:00–5:00

Teaching Assistants: Nate Foster
Office hours: Thurs 4:30-5:30 in GRW 565
Dimitrios Vytiniotis
Office hours: Tues 2:00-3:00 in GRW 565

Administrative Assistant

Cheryl Hickey, Levine 502

If you are unable to reach me please contact Cheryl Hickey, 215-898-3538 or `cherylh@central.cis.upenn.edu`. You may find your class folder in the filing cabinet outside of Room 502 Levine for all graded homeworks and extra handouts. Please see Cheryl for your graded exams.

Temporary Personnel

I will be away from September 10 to September 22.

Guest lecturers for next three lectures.

- Dr. Benjamin Pierce, September 13 and 15.
- Dr. Val Tannen, September 20.

Contact me using course email: `cis500@cis.upenn.edu`.

Information

Textbook: Types and Programming Languages,
Benjamin C. Pierce, MIT Press, 2002

Webpage: `http://www.seas.upenn.edu/~cis500`

Newsgroup: `upenn.cis.cis500`

Exams

1. First mid-term: Wed, October 13
2. Second mid-term: Wed, November 15
3. Final: TBA.

Additional administrative information will be posted as necessary during the semester. Keep an eye on the course web page and (especially) the newsgroup.

Final exam

The final for this course has been scheduled by the registrar for Thurs. 12/16/04, 8:30-10:30.

The final for CIS 501 has been scheduled by the registrar for Thurs. 12/16/04, 11-1

Final exam

The final for this course has been scheduled by the registrar for Thurs. 12/16/04, 8:30-10:30.

The final for CIS 501 has been scheduled by the registrar for Thurs. 12/16/04, 11-1

Exam can be rescheduled by a petition signed by every registered student. I'll pass the petition around after the add period is over (Friday, Sept 24).

If you have constraints, other than CIS courses, send them to `cis500@cis.upenn.edu`.

Grading

Final course grades will be computed as follows:

- Homework: 20%
- 2 midterms: 20% each
- Final: 40%

Extra Credit

Course grades can be improved after the semester ends in two ways:

1. A 1/3 letter grade improvement can be obtained by doing a substantial extra credit project (~30 hours work) during the Spring semester.
2. Larger grade improvements can (only) be obtained by sitting in on the course next year and turning in all homeworks and exams. If you are doing this to improve your grade from last year, let me know.

Collaboration

- Collaboration on homework is strongly encouraged
- Studying with other people is the best way to internalize the material
- Form study groups! 2 or 3 people is a nice size. 4 is too many for all to have equal input.
- We will help form groups for those that have not already done so
- Even if you are fairly confident about the course, you should be in a group.

"You never really misunderstand something until you try to teach it..."
— Anon.

Homework

- Readings from TAPL
  - Should be completed before lecture (see course web page).
  - Do all one star questions while reading (do not need to turn in).
  - Write down questions to ask in class or recitation.
- Written homework
  - Small part of your grade, yet a large part of your understanding.
  - Submit one assignment per study group. Submit all assignments this semester with the same group. You must form your study group before the first assignment is due. Even if you find this assignment easy, you will a group for later assignments!
  - Grading is random but fair. We may not grade every problem.
  - Some solutions are in the back of the book. Write your answer down before looking.
  - Late (non-)policy: Homework will not be accepted after the announced deadline.

First Homework Assignment

- The first homework assignment is due a week from Monday, September 20, by noon.
- The assignment is posted on the course web page.
- All assignments must be typeset and submitted electronically. The use of LaTeX is strongly encouraged.

Recitations

- Everyone in the class should attend one of the recitation sections
- There are two kinds of recitations:
  1. Review sections will focus on material close to what is presented in class and on homeworks
  2. Advanced sections will introduce additional related material
  3. Meetings of recitation sections will start next week, except for the advanced recitation.

Wed 3:30-5:00 PM	DRLB 4C2	advanced
Wed 3:30-5:00 PM	DRLB 4E9	review
Thurs 1:30-3 PM	Towne 321	review
Thurs 10:30-12 PM	Towne 307	review
Fri 9:30-11 AM	Towne 307	review

The WPE-I

- PhD students in CIS must pass a five-section Written Preliminary Exam (WPE-I)

  Software Foundations is one of the five areas
- The final for this course is also the software foundations WPE-I exam
- Near the end of the semester, you will be given an opportunity to declare your intention to take the final exam for WPE credit

The WPE-I (continued)

- You do not need to be enrolled in the course to take the exam for WPE credit
- If you are enrolled in the course and also take the exam for WPE credit, you will receive two grades: a letter grade for the course final and a Pass/Fail for the WPE
- You may take the exam for WPE credit even if you are not currently enrolled in the PhD program.

The WPE-I syllabus

- Reading knowledge of core OCaml
- Chapters 1-11 and 13-19 of TAPL

Announcement

- The department offers a Faculty Research Seminar most weeks during the Fall semester
- Friday afternoons, 3:30 – 4:30, in Levine Auditorium
- Speakers and topics are announced on the CIS newsgroups
- First-year CIS PhD students are required to attend. Others are welcome.

What is a programming language?

Syntax

Defining a programming language

We can define the terms of a programming language in a number of different ways.

Here is a BNF grammar for a very simple language of boolean expressions:

```
t ::=                          terms
      true                     constant true
      false                    constant false
      not t                    negation
      if t then t else t       conditional
```

Terminology:

- `t` here is a metavariable

Another form of the definition

The set $\mathcal{B}$ of boolean terms is the smallest set such that

1. $\{\texttt{true}, \texttt{false}\} \subseteq \mathcal{B}$;
2. if $\texttt{t}_1 \in \mathcal{B}$, then $\{\texttt{not t}_1\} \subseteq \mathcal{B}$;
3. if $\texttt{t}_1 \in \mathcal{B}$, $\texttt{t}_2 \in \mathcal{B}$, and $\texttt{t}_3 \in \mathcal{B}$, then $\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{B}$.

Abstract vs. concrete syntax

Q1: Does this grammar define a set of character strings, a set of token lists, or a set of abstract syntax trees?

Abstract vs. concrete syntax

Q1: Does this grammar define a set of character strings, a set of token lists, or a set of abstract syntax trees?

A: In a sense, all three. But we are interested in abstract syntax trees.

For this reason, grammars like the one on the previous slide are sometimes called abstract grammars. An abstract grammar defines a set of abstract syntax trees and suggests a mapping from character strings to trees.

We then write terms as linear character strings rather than trees simply for convenience. If there is any potential confusion about what tree is intended, we use parentheses to disambiguate.

Q: So, are

```
not false
not (false)
(((not (((((false)))))))
```

"the same term"?

What about

```
true
not false
```

?

Abstract Syntax, not semantics

We've only defined the abstract syntax of our language. That means our language is just a set of terms.

We haven't assigned any meanings to those terms yet. So there is no reason why we should equate `true` or `not false`, they're just uninterpreted terms.

Soon we will start talking about how we can decide what these terms mean.

Semantics

Defining what a language "means"

As well as defining the syntax of a programming language, we also need to define its semantics or the "meaning" of expressions written in that language.

Styles of semantics

1. Operational Semantics specifies the behavior of programs, much like an interpreter.
2. Denotational Semantics translates programs to a domain that we already know the meaning of: mathematics. The meaning of a term is a mathematical object like a function.
3. Axiomatic Semantics describes the meaning of a program through laws that describe its behavior.

In this course we will concentration on operational semantics.

Operational Semantics

- Describes the evaluation of programs on an abstract machine.
- Defined by a relation between each program and its result of evaluation.
- Several ways to define operational semantics—we'll look at a few in this course.
- We want the programs `not false` and `true` to mean the same thing.

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

- `(true, true)` ∈ ***Eval***

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

- `(true, true)` ∈ ***Eval***
- `(false, false)` ∈ ***Eval***

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

- `(true, true)` ∈ ***Eval***
- `(false, false)` ∈ ***Eval***
- `(not t, true)` ∈ ***Eval*** when `(t, false)` ∈ ***Eval***

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

- $(\texttt{true}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{false}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not t}, \texttt{true}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not t}, \texttt{false}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{true}) \in \textit{Eval}$

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

- $(\texttt{true}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{false}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not t}, \texttt{true}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not t}, \texttt{false}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3, \texttt{t}) \in \textit{Eval}$ when either:
  - $(t_1, \texttt{true}) \in \textit{Eval}$ and $(t_2, \texttt{t}) \in \textit{Eval}$
  - $(t_1, \texttt{false}) \in \textit{Eval}$ and $(t_3, \texttt{t}) \in \textit{Eval}$

Operational Semantics

Eval is a relation between terms in $\mathcal{B}$. It is the smallest set such that:

- $(\texttt{true}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{false}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not t}, \texttt{true}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not t}, \texttt{false}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3, \texttt{t}) \in \textit{Eval}$ when either:
  - $(t_1, \texttt{true}) \in \textit{Eval}$ and $(t_2, \texttt{t}) \in \textit{Eval}$
  - $(t_1, \texttt{false}) \in \textit{Eval}$ and $(t_3, \texttt{t}) \in \textit{Eval}$

If $(t_1, t_2) \in \textit{Eval}$ we say that t_2 is the meaning of t_1.

Properties of boolean language

Now that we have defined the syntax and semantics of the boolean language, what properties are true?

- $(\texttt{true}, \texttt{false}) \notin \textit{Eval}$.
- `not false` and `true` have the same meaning.
- All boolean terms have meanings (***Eval*** is total).
- There is only one meaning for each term (***Eval*** is deterministic).

How do we show that these properties are true?

Proving properties about programming languages

We want to show that a property is true for all $t \in \mathcal{B}$?

Can't do it by case analysis: $\mathcal{B}$ is an infinite set.

Example: Natural number induction

Principle of ordinary induction on natural numbers

Suppose that P is a predicate on the natural numbers. Then:

If $P(0)$
and, for all $i \in \mathcal{N}$, $P(i)$ implies $P(i+1)$,
then $P(n)$ holds for all $n \in \mathcal{N}$.

Natural numbers

The reason that we have an induction principle for natural numbers, is because they are defined in a certain way:

The set $\mathcal{N}$ is the smallest set such that

1. $0 \in \mathcal{N}$.
2. If $n \in \mathcal{N}$ then $n+1 \in \mathcal{N}$.

For shorthand, we sometimes abbreviate $0 + 1$ as 1, and $0 + 1 + 1$ as 2, and $0 + 1 + 1 + 1$ as 3, etc.

Example

Theorem: $2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$, for every n.

Proof:

- Let $P(i)$ be "$2^0 + 2^1 + \ldots + 2^i = 2^{i+1} - 1$."

Example

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n.

Proof:

- Let $P(i)$ be "$2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$."
- Show $P(0)$:

$$2^0 = 1 = 2^1 - 1$$

Example

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n.

Proof:

- Let $P(i)$ be "$2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$."
- Show $P(0)$:

$$2^0 = 1 = 2^1 - 1$$

- Show that $P(i)$ implies $P(i+1)$:

$$\begin{aligned} 2^0 + 2^1 + \dots + 2^{i+1} &= (2^0 + 2^1 + \dots + 2^i) + 2^{i+1} \\ &= (2^{i+1} - 1) + 2^{i+1} && \text{by IH} \\ &= 2 \cdot (2^{i+1}) - 1 \\ &= 2^{i+2} - 1 \end{aligned}$$

Example

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n.

Proof:

- Let $P(i)$ be "$2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$."
- Show $P(0)$:

$$2^0 = 1 = 2^1 - 1$$

- Show that $P(i)$ implies $P(i+1)$:

$$\begin{aligned} 2^0 + 2^1 + \dots + 2^{i+1} &= (2^0 + 2^1 + \dots + 2^i) + 2^{i+1} \\ &= (2^{i+1} - 1) + 2^{i+1} && \text{by IH} \\ &= 2 \cdot (2^{i+1}) - 1 \\ &= 2^{i+2} - 1 \end{aligned}$$

- The result ($P(n)$ for all n) follows by the principle of induction.

Shorthand form

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n.

Proof: By induction on n.

- Base case ($n = 0$): $2^0 = 1 = 2^1 - 1$
- Inductive case ($n = i + 1$):

$$\begin{aligned} 2^0 + 2^1 + \dots + 2^{i+1} &= (2^0 + 2^1 + \dots + 2^i) + 2^{i+1} \\ &= (2^{i+1} - 1) + 2^{i+1} && \text{IH} \\ &= 2 \cdot (2^{i+1}) - 1 \\ &= 2^{i+2} - 1 \end{aligned}$$

Inductive definitions

This is the same way we defined what boolean terms were.

The set $\mathcal{B}$ of boolean terms is the smallest set such that

1. $\{\texttt{true}, \texttt{false}\} \subseteq \mathcal{B}$;
2. if $\texttt{t}_1 \in \mathcal{B}$, then $\{\texttt{not}\ \texttt{t}_1\} \subseteq \mathcal{B}$;
3. if $\texttt{t}_1 \in \mathcal{B}$, $\texttt{t}_2 \in \mathcal{B}$, and $\texttt{t}_3 \in \mathcal{B}$, then $\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 \in \mathcal{B}$.

Structural Induction

We can also use induction for boolean terms. The way we have defined terms gives us an induction principle:

For all $\texttt{t} \in \mathcal{B}$, $P(\texttt{t})$ is true if and only if

- $P(\texttt{true})$ and $P(\texttt{false})$ hold
- for all $\texttt{t}_1 \in \mathcal{B}$, if $P(\texttt{t}_1)$ holds, then $P(\texttt{not}\ \texttt{t}_1)$ hold.
- for all $\texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in \mathcal{B}$, if $P(\texttt{t}_1)$, $P(\texttt{t}_2)$ and $P(\texttt{t}_3)$ holds, then $P(\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3)$ holds.

Proofs by induction

We'll prove that evaluation is deterministic. In other words: For all $\texttt{t}$ there exists at most one $\texttt{t}'$ such that $(\texttt{t}, \texttt{t}') \in \textit{Eval}$.

This gives us the property:

$P(\texttt{t})$ = exists at most one $\texttt{t}'$ such that $(\texttt{t}, \texttt{t}') \in \textit{Eval}$.

So we want to show:

- $P(\texttt{true})$ (i.e. exists at most one $\texttt{t}'$ such that $(\texttt{true}, \texttt{t}') \in \textit{Eval}$)
- $P(\texttt{false})$
- $P(\texttt{not}\ \texttt{t}_1)$ given that $P(\texttt{t}_1)$ holds.
- $P(\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3)$ given that $P(\texttt{t}_1)$, $P(\texttt{t}_2)$ and $P(\texttt{t}_3)$ all hold.

Definition of *Eval*

Definition: *Eval* is the smallest set such that:

- $(\texttt{true}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{false}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not}\ \texttt{t}, \texttt{true}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{false}) \in \textit{Eval}$
- $(\texttt{not}\ \texttt{t}, \texttt{false}) \in \textit{Eval}$ when $(\texttt{t}, \texttt{true}) \in \textit{Eval}$
- $(\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3, \texttt{t}) \in \textit{Eval}$ when either:
  - $(\texttt{t}_1, \texttt{true}) \in \textit{Eval}$ and $(\texttt{t}_2, \texttt{t}) \in \textit{Eval}$
  - $(\texttt{t}_1, \texttt{false}) \in \textit{Eval}$ and $(\texttt{t}_3, \texttt{t}) \in \textit{Eval}$