CS 6110 — Advanced Programming Languages

Lecture 1
Introduction

24 January 2011

Programming Languages

One of the oldest fields in Computer Science...

- λ-calculus – Church (1936)
- FORTRAN – Backus (1957)
- LISP – McCarthy (1958)
- ALGOL 60 – Backus, Naur, Perlis, & others (1960)
- Pascal – Wirth (1970)
- C – Ritchie (1972)
- Smalltalk – Kay & others (1972)
- ML – Milner and others (1978)
- C++ – Stroustrup (1982)
- Haskell – Hudak, Peyton Jones, Wadler, & others (1989)
- Java – Gosling (1995)
- C# – Microsoft (2001)
- Scala – Odersky (2003)
- F# – Syme (2005)

Programming Languages

...and one of the most vibrant areas today!

PL intersects with many other areas

Current trends

- Domain-specific languages
- Static analysis and types
- Language-based security
- Verification and model checking
- Concurrency

Both theoretically and practically “meaty”

Syllabus

Course Goals

- Learn techniques for modeling programs*
  - Formal semantics (operational, axiomatic, denotational)
  - Extend to advanced language features
  - Develop reasoning principles (induction, co-induction)
- Explore applications of these techniques
  - Optimization
  - Static analysis
  - Verification
- PhD students: cover material for PL qualifying exam
- Have fun :-)

*and whole languages!

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of Computer Science

24 Jan	Introduction	PDF		
26 Jan	λ-calculus			
28 Jan	λ-calculus encodings and recursion			
31 Jan	Normal forms, reduction strategies, and confluence			
2 Feb	Substitution, big step vs. small-step			
4 Feb	Structured Operational Semantics and IMP			
7 Feb	Inductive definitions and least fixed points			
9 Feb	Well-Founded Induction and rule induction			
11 Feb	Evaluation contexts and definitional Translation			
14 Feb	uML and strong typing			
16 Feb	Naming and scope			
18 Feb	Recursive bindings and modules			
21 Feb	State and mutable variables			
23 Feb	Call by reference, continuation-passing style, CPS conversion			
25 Feb	Non-local control, errors, and exceptions			
28 Feb	First-class continuations and threads			
2 Mar	Compiling with continuations			
4 Mar	Hoare logic			
7 Mar	Weakest preconditions			
9 Mar	Verification conditions and applications			
11 Mar	Denotational semantics of IMP			
14 Mar	The fixed-point theorem			
16 Mar	Domain constructions			
18 Mar	Metalanguage for denotational semantics			

21 Mar	Spring break (no class)			
23 Mar	Spring break (no class)			
25 Mar	Spring break (no class)			
28 Mar	Review			
30 Mar	Simply-typed λ-calculus			
1 Apr	Products, sums, and more			
4 Apr	Type soundness			
6 Apr	Subtyping			
8 Apr	Minimal typing			
11 Apr	Type inference			
13 Apr	Parametric polymorphism			
15 Apr	Strong normalization and logical relations			
18 Apr	Propositions as types			
20 Apr	Recursive types			
22 Apr	Solving recursive domain equations			
25 Apr	Existential types			
27 Apr	Parameterized types			
29 Apr	Bounded quantification			
2 May	Object encodings			
4 May	Current research in Programming Languages			
6 May	Review			
9 May	Study Period (no class)			
11 May	Study Period (no class)			
13 May	Final Exam (2:00-4:30pm)			

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of
Computer Science

Mathemetical Preliminaries & Operational Semantics

24 Jan	Introduction	PDF		
26 Jan	λ-calculus			
28 Jan	λ-calculus encodings and recursion			
31 Jan	No[illegible]			
2 Feb	Substitution, big step vs. small-step			
4 Feb	Structured Op[illegible] Semantics and IMP			
7 Feb	Inductive definitions and least fixed points			
9 Feb	Well-Founded Induction and rule induction			
11 Feb	Evaluation contexts and definitional Translation			
14 Feb	uML and strong typing			
16 Feb	Naming and scope			
18 Feb	Recursive bindings and modules			
21 Feb	State and mutable variables			
23 Feb	Call by reference, continuation-passing style, CPS conversion			
25 Feb	Non-local control, errors, and exceptions			
28 Feb	First-class continuations and threads			
2 Mar	Compiling with continuations			
4 Mar	Hoare logic			
7 Mar	Weakest preconditions			
9 Mar	Verification conditions and applications			
11 Mar	Denotational semantics of IMP			
14 Mar	The fixed-point theorem			
16 Mar	Domain constructions			
18 Mar	Metalanguage for denotational semantics			

21 Mar	Spring break (no class)			
23 Mar	Spring break (no class)			
25 Mar	Spring break (no class)			
28 Mar	Review			
30 Mar	Simply-typed λ-calculus			
1 Apr	Products, sums, and more			
4 Apr	Type soundness			
6 Apr	Subtyping			
8 Apr	Minimal typing			
11 Apr	Type inference			
13 Apr	Parametric polymorphism			
15 Apr	Strong normalization and logical relations			
18 Apr	Propositions as types			
20 Apr	Recursive types			
22 Apr	Solving recursive domain equations			
25 Apr	Existential types			
27 Apr	Parameterized types			
29 Apr	Bounded quantification			
2 May	Object encodings			
4 May	Current research in Programming Languages			
6 May	Review			
9 May	Study Period (no class)			
11 May	Study Period (no class)			
13 May	Final Exam (2:00-4:30pm)			

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of Computer Science

Home Syllabus Schedule Resources

Mathemetical Preliminaries & Operational Semantics

Advanced Language Features

24 Jan	Introduction	PDF		
26 Jan	λ-calculus			
28 Jan	λ-calculus encodings and recursion			
31 Jan	[illegible]			
2 Feb	Substitution, big step vs. small step			
4 Feb	Structured Op[illegible]			
7 Feb	Inductive definitions and least fixed points			
9 Feb	Well-Founded Induction and rule induction			
11 Feb	Evaluation contexts and definitional Translation			
14 Feb	uML and strong typing			
16 Feb	Naming and scope			
18 Feb	Recursive bindings and modules			
21 Feb	State and mutable variables			
23 Feb	Call by reference, continuation-passing style, CPS conversion			
25 Feb	Non-local control, errors, and exceptions			
28 Feb	First-class continuations and threads			
2 Mar	Compiling with continuations			
4 Mar	Hoare logic			
7 Mar	Weakest preconditions			
9 Mar	Verification conditions and applications			
11 Mar	Denotational semantics of IMP			
14 Mar	The fixed-point theorem			
16 Mar	Domain constructions			
18 Mar	Metalanguage for denotational semantics			

21 Mar	Spring break (no class)			
23 Mar	Spring break (no class)			
25 Mar	Spring break (no class)			
28 Mar	Review			
30 Mar	Simply-typed λ-calculus			
1 Apr	Products, sums, and more			
4 Apr	Type soundness			
6 Apr	Subtyping			
8 Apr	Minimal typing			
11 Apr	Type inference			
13 Apr	Parametric polymorphism			
15 Apr	Strong normalization and logical relations			
18 Apr	Propositions as types			
20 Apr	Recursive types			
22 Apr	Solving recursive domain equations			
25 Apr	Existential types			
27 Apr	Parameterized types			
29 Apr	Bounded quantification			
2 May	Object encodings			
4 May	Current research in Programming Languages			
6 May	Review			
9 May	Study Period (no class)			
11 May	Study Period (no class)			
13 May	Final Exam (2:00-4:30pm)			

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of
Computer Science

Mathemetical Preliminaries & Operational Semantics

24 Jan	Introduction	PDF
26 Jan	λ-calculus	
28 Jan	λ-calculus encodings and recursion	
31 Jan		
2 Feb	Substitution	
4 Feb	Structured	
7 Feb	Inductive definitions and least fixed points	
9 Feb	Well-Founded Induction and rule induction	
11 Feb	Evaluation contexts and definitional Translation	

Advanced Language Features

14 Feb	uML and strong typing
16 Feb	Naming and scope
18 Feb	Recursive bindings and modules
21 Feb	
23 Feb	
25 Feb	Non-local control, errors, and exceptions
28 Feb	First-class continuations and threads
2 Mar	Compiling with continuations
4 Mar	Hoare logic

Axiomatic & Denotational Semantics

7 Mar	Weakest preconditions
9 Mar	
11 Mar	Denotational semantics of IMP
14 Mar	The fixed-point theorem
16 Mar	Domain constructions
18 Mar	Metalanguage for denotational semantics

21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)
28 Mar	Review
30 Mar	Simply-typed λ-calculus
1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types
22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review
9 May	Study Period (no class)
11 May	Study Period (no class)
13 May	Final Exam (2:00-4:30pm)

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of
Computer Science

Home Syllabus Schedule Resources

Mathemetical Preliminaries & Operational Semantics

Advanced Language Features

Axiomatic & Denotational Semantics

Spring Break

24 Jan	Introduction	PDF
26 Jan	λ-calculus	
28 Jan	λ-calculus encodings and recursion	
31 Jan	No[illegible]	
2 Feb	Substitution, [illegible] step vs. small step	
4 Feb	Structured Op[illegible]	
7 Feb	Inductive definitions and least fixed points	
9 Feb	Well-Founded Induction and rule induction	
11 Feb	Evaluation contexts and definitional Translation	
14 Feb	uML and strong typing	
16 Feb	Naming and scope	
18 Feb	Recursive bindings and modules	
21 Feb	State and mutable variables	
23 Feb	Call by [illegible]	
25 Feb	Non-local control, errors, and exceptions	
28 Feb	First-class continuations and threads	
2 Mar	Compiling with continuations	
4 Mar	Hoare logic	
7 Mar	Weakest preconditions	
9 Mar	Verificati[illegible]	
11 Mar	Denotational semantics of IMP	
14 Mar	The fixed-point theorem	
16 Mar	Domain constructions	
18 Mar	Metalanguage for denotational semantics	

21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)
28 Mar	Review
30 Mar	Simply-typed λ-calculus
1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types
22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review
9 May	Study Period (no class)
11 May	Study Period (no class)
13 May	Final Exam (2:00-4:30pm)

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of Computer Science

Home Syllabus Schedule Resources

Mathemetical Preliminaries & Operational Semantics

24 Jan	Introduction	PDF
26 Jan	λ-calculus	
28 Jan	λ-calculus encodings and recursion	
31 Jan	[illegible]	
2 Feb	[illegible]	
4 Feb	[illegible]	
7 Feb	Inductive definitions and least fixed points	
9 Feb	Well-Founded Induction and rule induction	
11 Feb	Evaluation contexts and definitional Translation	

Advanced Language Features

14 Feb	uML and strong typing	
16 Feb	Naming and scope	
18 Feb	Recursive bindings and modules	
21 Feb	[illegible]	
23 Feb	[illegible]	
25 Feb	Non-local control, errors, and exceptions	
28 Feb	First-class continuations and threads	
2 Mar	Compiling with continuations	
4 Mar	Hoare logic	

Axiomatic & Denotational Semantics

7 Mar	Weakest preconditions	
9 Mar	[illegible]	
11 Mar	Denotational semantics of IMP	
14 Mar	The fixed-point theorem	
16 Mar	Domain constructions	
18 Mar	Metalanguage for denotational semantics	

Spring Break

21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)

Preliminary Exam

28 Mar	Review
30 Mar	Simply-typed λ-calculus
1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types
22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review
9 May	Study Period (no class)
11 May	Study Period (no class)
13 May	Final Exam (2:00-4:30pm)

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of
Computer Science

Home Syllabus Schedule Resources

Mathemetical Preliminaries & Operational Semantics

24 Jan	Introduction	PDF
26 Jan	λ-calculus	
28 Jan	λ-calculus encodings and recursion	
31 Jan		
2 Feb		
4 Feb		
7 Feb	Inductive definitions and least fixed points	
9 Feb	Well-Founded Induction and rule induction	
11 Feb	Evaluation contexts and definitional Translation	

Advanced Language Features

14 Feb	uML and strong typing
16 Feb	Naming and scope
18 Feb	Recursive bindings and modules
21 Feb	
23 Feb	
25 Feb	Non-local control, errors, and exceptions
28 Feb	First-class continuations and threads
2 Mar	Compiling with continuations
4 Mar	Hoare logic

Axiomatic & Denotational Semantics

7 Mar	Weakest preconditions
9 Mar	
11 Mar	
14 Mar	The fixed-point theorem
16 Mar	Domain constructions
18 Mar	Metalanguage for denotational semantics

Spring Break

21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)

Preliminary Exam

28 Mar	Review
30 Mar	Simply-typed λ-calculus

Types

1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types

22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review
9 May	Study Period (no class)
11 May	Study Period (no class)
13 May	Final Exam (2:00-4:30pm)

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of Computer Science

Home | Syllabus | Schedule | Resources

Mathemetical Preliminaries & Operational Semantics

24 Jan	Introduction	PDF
26 Jan	λ-calculus	
28 Jan	λ-calculus encodings and recursion	
31 Jan		
2 Feb	Substitution	
4 Feb	Structured	
7 Feb	Inductive definitions and least fixed points	
9 Feb	Well-Founded Induction and rule induction	
11 Feb	Evaluation contexts and definitional Translation	

Advanced Language Features

14 Feb	uML and strong typing
16 Feb	Naming and scope
18 Feb	Recursive bindings and modules
21 Feb	
23 Feb	
25 Feb	Non-local control, errors, and exceptions
28 Feb	First-class continuations and threads
2 Mar	Compiling with continuations
4 Mar	Hoare logic

Axiomatic & Denotational Semantics

7 Mar	Weakest preconditions
9 Mar	Verificat
11 Mar	Denotational semantics of IMP
14 Mar	The fixed-point theorem
16 Mar	Domain constructions
18 Mar	Metalanguage for denotational semantics

Spring Break

21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)

Preliminary Exam

28 Mar	Review
30 Mar	Simply-typed λ-calculus

Types

1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types

Advanced Types

22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review

9 May	Study Period (no class)
11 May	Study Period (no class)
13 May	Final Exam (2:00-4:30pm)

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of Computer Science

Mathematical Preliminaries & Operational Semantics

24 Jan	Introduction	PDF
26 Jan	λ-calculus	
28 Jan	λ-calculus encodings and recursion	
31 Jan	No[illegible]	
2 Feb	Substitution, big step vs. small step	
4 Feb	Structured Op[illegible]	
7 Feb	Inductive definitions and least fixed points	
9 Feb	Well-Founded Induction and rule induction	
11 Feb	Evaluation contexts and definitional Translation	

Advanced Language Features

14 Feb	uML and strong typing
16 Feb	Naming and scope
18 Feb	Recursive bindings and modules
21 Feb	State and mutable variables
23 Feb	Call-by-reference, [illegible]
25 Feb	Non-local control, errors, and exceptions
28 Feb	First-class continuations and threads
2 Mar	Compiling with continuations
4 Mar	Hoare logic

Axiomatic & Denotational Semantics

7 Mar	Weakest preconditions
9 Mar	Verificati[illegible]
11 Mar	Denotational semantics of IMP
14 Mar	The fixed-point theorem
16 Mar	Domain constructions
18 Mar	Metalanguage for denotational semantics

Spring Break

21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)

Preliminary Exam

28 Mar	Review
30 Mar	Simply-typed λ-calculus

Types

1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types

Advanced Types

22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review

Study Period

9 May	Study Period (no class)
11 May	Study Period (no class)

Final Exam

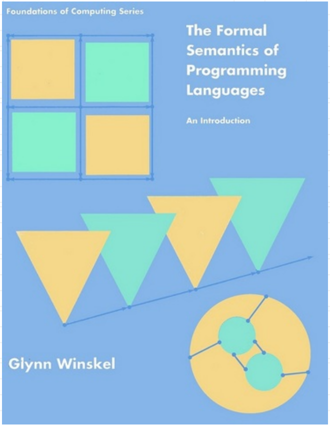
13 May	Final Exam (2:00-4:30pm)

CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of
Computer Science

24 Jan	
26 Jan	
28 Jan	
31 Jan	
2 Feb	
4 Feb	
7 Feb	
9 Feb	
11 Feb	
14 Feb	
16 Feb	
18 Feb	
21 Feb	
23 Feb	
25 Feb	
28 Feb	
2 Mar	
4 Mar	
7 Mar	
9 Mar	
11 Mar	
14 Mar	
16 Mar	
18 Mar	



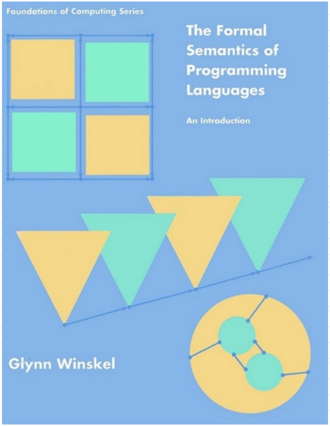
21 Mar	Spring break (no class)
23 Mar	Spring break (no class)
25 Mar	Spring break (no class)
28 Mar	Review
30 Mar	Simply-typed λ-calculus
1 Apr	Products, sums, and more
4 Apr	Type soundness
6 Apr	Subtyping
8 Apr	Minimal typing
11 Apr	Type inference
13 Apr	Parametric polymorphism
15 Apr	Strong normalization and logical relations
18 Apr	Propositions as types
20 Apr	Recursive types
22 Apr	Solving recursive domain equations
25 Apr	Existential types
27 Apr	Parameterized types
29 Apr	Bounded quantification
2 May	Object encodings
4 May	Current research in Programming Languages
6 May	Review
9 May	Study Period (no class)
11 May	Study Period (no class)
13 May	Final Exam (2:00-4:30pm)

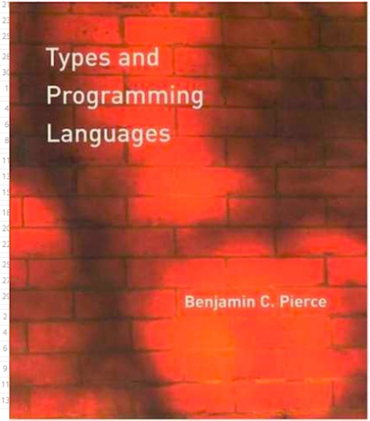
CS 6110 (Spring 2011)

Advanced Programming Languages
MWF 10:10-11:00
Phillips 213

Cornell University
Department of
Computer Science

Home Syllabus Schedule Resources





Prerequisites

Programming Experience

- e.g., C, Java, Prolog, OCaml, Haskell, Scheme/Racket
- Comfortable with a functional language
- For undergrads: CS 3110 or 4110 or equivalent

Mathematical Maturity

- e.g., set theory, rigorous proofs, induction
- Much of this class will involve formal reasoning
- Hardest topic: denotational semantics

Interest (having fun is a goal! :-)

If you don’t meet these prerequisites, get in touch.

Course Work

Participation (5%)

- Lectures, recitations, and office hours
- Email list discussions

Homework (25%)

- 6 assignments, roughly every other week
- Mostly theoretical, some programming
- *Must* work in groups of 2-3

Preliminary Exam (30%)

- Wednesday, March 30th + take-home problems.

Final Exam (40%)

- Friday, May 13th, 2pm-4:30pm
- Cumulative, with focus on the material from 2nd half

Academic Integrity

Two simple requests:

1. Most of you are here training to become members of the research community. Conduct yourself with integrity.

2. If you aren't sure what is allowed and what isn't, please ask!

Special Needs and Wellness

- I will provide reasonable accommodations to students who have a documented disability (e.g., physical, learning, psychiatric, vision, hearing, or systemic).

- If you are experiencing undue personal or academic stress at any time during the semester (or if you notice that a fellow student is), contact me, Engineering Advising, or Gannett.

Course Staff

Instructor

Nate Foster
Office: Upson 4137
Hours: Wed 11am-12pm

Teaching Assistant

Jean-Baptiste Jeannin
Office: Upson 4142
Hours: Tue 4:45pm-5:45pm and Thu 7pm-8pm

(office hours start next week)

Web Page

`http://www.cs.cornell.edu/Courses/cs6110/2011sp`

Mailing List

`http://lists.semantics-is-gorges.org/listinfo/cs6110`

Language Specification

Language Specification

Formal Semantics: what do programs mean?

Three Approaches

- Operational
  - Models program by its execution on abstract machine
  - Useful for implementing compilers and interpreters
- Axiomatic
  - Models program by the logical formulas it obeys
  - Useful for proving program correctness
- Denotational
  - Models program literally as mathematical objects
  - Useful for theoretical foundations

Language Specification

Formal Semantics: what do programs mean?

Three Approaches

- Operational
  - Models program by its execution on abstract machine
  - Useful for implementing compilers and interpreters
- Axiomatic
  - Models program by the logical formulas it obeys
  - Useful for proving program correctness
- Denotational
  - Models program literally as mathematical objects
  - Useful for theoretical foundations

Question: few languages have a formal semantics. Why?

Formal Semantics

Too Hard?

- Modeling a real-world language is hard
- Notation can gets very dense
- Sometimes requires developing new mathematics
- Not yet cost-effective for everyday use

Overly General?

- Explains the behavior of a program on *every* input
- Most programmers are content knowing the behavior of their program on *this* input (or these inputs)

Okay, so who needs semantics?

A Tricky Example

Question #1: is the following Java program legal?

Question #2: if yes, what does it do?

```
class A { static int a = B.b + 1; }
class B { static int b = A.a + 1; }
```

Who Needs Semantics?

Unambiguous Description

- Anyone who wants to design a new feature
- Basis for most formal arguments
- Standard tool in PL research

Exhaustive Reasoning

- Sometimes have to know behavior on all inputs
- Compilers and interpreters
- Static analysis tools
- Program transformation tools
- Critical software

Language Design

Design Desiderata

Question: What makes a good programming language?

Design Desiderata

Question: What makes a good programming language?

One answer: "a good language is one people use"

Design Desiderata

Question: What makes a good programming language?

One answer: "a good language is one people use"

Wrong! Are COBOL and JavaScript the best languages?

Design Desiderata

Question: What makes a good programming language?

One answer: “a good language is one people use”

Wrong! Are COBOL and JavaScript the best languages?

Some good features:

- Simplicity (clean, orthogonal constructs)
- Readability (elegant syntax)
- Safety (guarantees that programs won’t “go wrong”)
- Support for programming in the large (modularity)
- Efficiency (good execution model and tools)

Design Challenges

Unfortunately these goals almost always conflict

- Types restrict expressiveness in general, but they provide strong guarantees
- Safety checks eliminate errors but have a cost, either when compiling or when the program is executed
- Some verification tools are so complicated, one essentially needs a PhD to use them

Design Challenges

Unfortunately these goals almost always conflict

- Types restrict expressiveness in general, but they provide strong guarantees
- Safety checks eliminate errors but have a cost, either when compiling or when the program is executed
- Some verification tools are so complicated, one essentially needs a PhD to use them

A lot of PL research is about finding ways to gain without too much pain

Story: Unexpected Interactions

A real story illustrating the perils of language design

Cast of characters includes famous computer scientists

Timeline:

- 1982: ML is a functional language with type inference, polymorphism (generics), and monomorphic references (pointers)
- 1985: Standard ML innovates by adding polymorphic references $\rightarrow$ unsoundness
- 1995: The “innovation” fixed

ML Type System

Polymorphism: allows code to be used at different types

Examples:

- List.length : $\forall\alpha.\ \alpha$ list $\rightarrow$ int
- List.hd : $\forall\alpha.\ \alpha$ list $\rightarrow \alpha$

Type Inference: $e \rightsquigarrow \tau$

- e.g., let *id* (*x*) $=$ *x* $\rightsquigarrow \forall\alpha.\ \alpha \rightarrow \alpha$
- Generalize types not constrainted by the program
- Instantiate types at use *id* (true) $\rightsquigarrow$ bool

ML References

By default, values in ML are immutable.

But can extend the language with imperative features.

Add reference types of the form τ ref

Add expressions of the form

- ref *e* : τ ref where *e* : τ (allocate)
- !e : τ where *e* : τ ref (dereference)
- $e_1 := e_2$: unit where $e_1 : \tau$ *ref* and $e_2 : \tau$ (assign)

Works as you'd expect—i.e., just like pointers in C

Polymorphism + References

Consider the following program

Code	Inferred Type
let id(x) = x	id : $\forall\alpha\ \alpha \rightarrow \alpha$

Polymorphism + References

Consider the following program

Code	Inferred Type
let id(x) = x	id : $\forall\alpha\ \alpha \to \alpha$
let p = ref id	p : $\forall\alpha\ (\alpha \to \alpha)$ ref

Polymorphism + References

Consider the following program

Code	Inferred Type
let id(x) = x	id : $\forall\alpha\ \alpha \rightarrow \alpha$
let p = ref id	p : $\forall\alpha\ (\alpha \rightarrow \alpha)$ ref
let inc(n) = n+1	inc : int $\rightarrow$ int

Polymorphism + References

Consider the following program

Code	Inferred Type
let id(x) = x	id : $\forall\alpha\ \alpha \to \alpha$
let p = ref id	p : $\forall\alpha\ (\alpha \to \alpha)$ ref
let inc(n) = n+1	inc : int $\to$ int
let () = p := inc	OK since p : (int $\to$ int) ref

Polymorphism + References

Consider the following program

Code	Inferred Type
let id(x) = x	id : $\forall\alpha\ \alpha \to \alpha$
let p = ref id	p : $\forall\alpha\ (\alpha \to \alpha)$ ref
let inc(n) = n+1	inc : int $\to$ int
let () = p := inc	OK since p : (int $\to$ int) ref
(!p) true	OK since p : (bool $\to$ bool) ref

Polymorphism + References

Problem

- Type system is not sound
- Well-typed program $\longrightarrow^*$ type error!

Polymorphism + References

Problem

- Type system is not sound
- Well-typed program $\longrightarrow^*$ type error!

Proposed Solutions

1. "Weak" type variables
   - Can only be instantiated in restricted ways
   - But type exposes functional vs. imperative
   - Somewhat difficult to use

Polymorphism + References

Problem

- Type system is not sound
- Well-typed program $\longrightarrow^*$ type error!

Proposed Solutions

1. "Weak" type variables
   - Can only be instantiated in restricted ways
   - But type exposes functional vs. imperative
   - Somewhat difficult to use
2. Value restriction
   - Only generalize types of values
   - Most ML programs already obey it
   - Simple proof of type soundness

Lessons Learned

- Features often interact in unexpected ways
- The design space is huge
- Good designs are sparse $\rightarrow$ don't happen by accident
- Simplicity is rare: n features lead to n^2 interactions
- Most PL researchers work with really small languages (e.g., λ-calculus) to study core issues in isolation
- But must pay attention to whole languages too

Mathematical Preliminaries

Binary Relations

The *product* of two sets A and B, written $A \times B$, contains all ordered pairs (a, b) with $a \in A$ and $b \in B$.

Binary Relations

The *product* of two sets A and B, written $A \times B$, contains all ordered pairs (a, b) with $a \in A$ and $b \in B$.

A *binary relation* on A and B is just a subset $R \subseteq A \times B$.

Binary Relations

The *product* of two sets A and B, written $A \times B$, contains all ordered pairs (a, b) with $a \in A$ and $b \in B$.

A *binary relation* on A and B is just a subset $R \subseteq A \times B$.

Given a binary relation $R \subseteq A \times B$, the set A is called the *domain* of R and B is called the *range* (or *codomain*) of R.

Binary Relations

The *product* of two sets A and B, written $A \times B$, contains all ordered pairs (a, b) with $a \in A$ and $b \in B$.

A *binary relation* on A and B is just a subset $R \subseteq A \times B$.

Given a binary relation $R \subseteq A \times B$, the set A is called the *domain* of R and B is called the *range* (or *codomain*) of R.

Some Important Relations

- empty – $\emptyset$
- total – $A \times B$
- identity on A – $\{(a, a) \mid a \in A\}$.
- composition $R; S$ – $\{(a, c) \mid \exists b.\, (a, b) \in R \wedge (b, c) \in S\}$

Functions

A (*total*) *function* f is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

Functions

A (*total*) *function* f is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When f is a function, we usually write $f : A \rightarrow B$ instead of $f \subseteq A \times B$.

Functions

A (*total*) *function* f is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When f is a function, we usually write $f : A \rightarrow B$ instead of $f \subseteq A \times B$.

The *domain* and *range* of f are defined in exactly the same way as for relations.

Functions

A (*total*) *function* f is a binary relation $f \subseteq A \times B$ with the property that every $a \in A$ is related to exactly one $b \in B$.

When f is a function, we usually write $f : A \to B$ instead of $f \subseteq A \times B$.

The *domain* and *range* of f are defined in exactly the same way as for relations.

The *image* of f is the set of elements $b \in B$ that are mapped to by at least one $a \in A$:

$$\{f(a) \mid a \in A\}$$

Some Important Functions

Given two functions $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition of f and g is defined by:

$$(g \circ f)(x) = g(f(x))$$

Note order!

Some Important Functions

Given two functions $f : A \to B$ and $g : B \to C$, the composition of f and g is defined by:

$$(g \circ f)(x) = g(f(x))$$

Note order!

A partial function $f : A \rightharpoonup B$ is a total function $f : A' \to B$ on a set $A' \subseteq A$. The notation $\mathrm{dom}(f)$ refers to A'.

Some Important Functions

Given two functions $f : A \to B$ and $g : B \to C$, the composition of f and g is defined by:

$$(g \circ f)(x) = g(f(x))$$

Note order!

A partial function $f : A \rightharpoonup B$ is a total function $f : A' \to B$ on a set $A' \subseteq A$. The notation $\mathrm{dom}(f)$ refers to A'.

A function $f : A \to B$ is said to be *injective* (or *one-to-one*) if and only if $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$.

Some Important Functions

Given two functions $f : A \rightarrow B$ and $g : B \rightarrow C$, the composition of f and g is defined by:

$$(g \circ f)(x) = g(f(x))$$

Note order!

A partial function $f : A \rightharpoonup B$ is a total function $f : A' \rightarrow B$ on a set $A' \subseteq A$. The notation $\mathrm{dom}(f)$ refers to A'.

A function $f : A \rightarrow B$ is said to be *injective* (or *one-to-one*) if and only if $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$.

A function $f : A \rightarrow B$ is said to be *surjective* (or *onto*) if and only if the image of f is B.

Extensions vs. Intensions

Mathematically, a function *f* is defined by its *extension*: the set of pairs of inputs and outputs.

A function can also be described by an *intensional* representation: a program or procedure that computes an output given an input.

Extensions vs. Intensions

Mathematically, a function *f* is defined by its *extension*: the set of pairs of inputs and outputs.

A function can also be described by an *intensional* representation: a program or procedure that computes an output given an input.

The same function can have several intensional representations—e.g., for the identity:

- $\lambda a.\, a$
- $\lambda a.$ if true then a else a
- $\lambda a.$ if false then a else a
- $\lambda a.\, \pi_1\ (a, a)$
- $\lambda a.\, \pi_2\ (a, a)$
- $\lambda a.\, (\lambda y.\, y)\ a$