CS 430/530
Formal Semantics

Zhong Shao
Yale University
Department of Computer Science

Course Overview
August 31, 2011
Today’s Lecture

• Why you should study formal semantics.
• How I intend to teach this course.
• Math background and predicate logic
What Is Formal Semantics?

formal --- “mathematically rigorous”

semantics --- “study of meanings”

Obviously:

• What is a programming language? What is a program?
• What are the meanings of specific language features and how they interact?
• How to make sure that a program behaves according to its “specification”?

But also:

• How do we explain these “meanings”? in which “language”?
• What is a meta logic? What is a mechanized meta logic?
• What is a specification language? What is its “semantics”??
Why Take CS-430?

• Software reliability and security are the biggest problems faced by the IT industry today! You are likely to worry about them in your future jobs.

• It will give you an edge over your competitors: industry and most other schools don’t teach this.

• It will improve your programming skills – because you will have a better appreciation of what your programs actually mean.

• You will be better able to compare and contrast programming languages, or even design your own.

• It is an important and exciting area of research, with many new ideas and perspectives frequently emerging.
Arianne 5

On June 4, 1996, the Arianne 5 took off on its maiden flight.

40 seconds into its flight it veered off course and exploded.

It was later found to be an error in reuse of a software component.

(This picture became quite popular in talks on software reliability and related topics.)
“Better, Faster, Cheaper”

In 1999, NASA lost both the Mars Polar Lander and the Climate Orbiter.

Later investigations determined software errors were to blame.

- Orbiter: Component reuse error.
- Lander: Precondition violation.
“After a crew member mistakenly entered a zero into the data field of an application, the computer system proceeded to divide another quantity by that zero. The operation caused a buffer overflow, in which data leaked from a temporary storage space in memory, and the error eventually brought down the ship's propulsion system. The result: the Yorktown was dead in the water for more than two hours.”
Therac-25

From 1985-1987, several cancer patients were killed or seriously injured as a result of being over-radiated by Therac-25, a radiation treatment facility.

The problem was due to a subtle race condition between concurrent processes.
Computer Viruses

Need we say more?

For more “software horror stories”, see http://www.cs.tau.ac.il/~nachumd/horror.html
Observations

• Failure often due to simple problems “in the details”.

• Small theorems about large programs would be useful.

• Need clearly specified interfaces and checking of interface compliance.

• Better languages would help!
Challenges

The impact and cost of software failures will increase, as will the demand for extensibility.

The distinction between “safety-critical” and “consumer electronics” software will fade away.

*Who will provide the technology for “safe” software systems?*
10 Emerging Technologies 2011,
In: Technology Review, 2011(6), MIT Press
http://www.technologyreview.com/tr10/

- Social Indexing: Facebook remaps the Web to personalize online services
- Homomorphic Encryption: Making cloud computing more secure
- Smart Transformers: Controlling the flow of electricity to stabilize the grid
- Cloud Streaming: Bringing high-performance software to mobile devices
- Gestural Interfaces: Controlling computers with our bodies
- Crash-Proof Code: Making critical software safer
- Cancer Genomics: Deciphering the genetics behind the disease
- Separating Chromosomes: A more precise way to read DNA will change how we treat disease
- Solid-State Batteries: High-energy cells for cheaper electric cars
- Synthetic Cells: Designing new genomes could speed the creation of vaccines and biofuel-producing bacteria
Opportunities

High assurance / reliability depends fundamentally on our ability to reason about programs.

The opportunities for new languages as well as formal semantics, type theory, computational logic, and so on, are great.
Certifying a computing host?

Need to reason about:

• human behaviors
• cosmic rays + natural disasters
• hardware failure
• software

VIEW #1: bug-free host impossible. Treat it as a biological system.
Certifying a computing host?

Need to reason about:

- human behaviors
- cosmic rays + natural disasters
- hardware failure
- software

VIEW #2: focus on software since it is a rigorous mathematical entity!
Certified software

Find a mathematical **proof** showing that

*if the HW/Env follows its model, the software will run according to its specification*
Certified OS kernel

Research tasks & key innovations:

• **new OS kernel** that can “crash-proof” the entire system & application SW

• **new PLs** for writing certified kernel plug-ins (new OCAP + DSPLs)

• **new formal methods** for automating proofs & specs (VeriML)
Certified “hypervisor” kernel

Problems w. existing platforms

- Attacks: Zero-Day Kernel Vulnerabilities (ZDKVs) & rogue driver certificates
- leads to rogue kernels
- leads to rogue WinCC/Step7 apps
- leads to rogue PLC firmware

New CRASH technologies

- A small certified “hypervisor” kernel provides a reliable ZDKV-free core to fall back on, even under attacks
- Information-Flow-Control to enforce security
- Mechanized proof certificates are unforgeable
Course Overview
My Goals

I have two goals:

• To teach the most common methods for specifying formal semantics. In particular, the denotational, operational, axiomatic and type-theoretic methods.

  This will give you the necessary tools to understand semantic specifications and to develop new ones.

• To survey existing language features to provide a deep understanding of what these features really mean, what they do, and how they compare.

  This will enable you to better evaluate existing languages and new ones as they are developed.
Prerequisites

• CS-201, CS-202, CS-223, CS-323 (or equivalents)

• Mathematical background: logic, sets, relations, functions, products, and unions. (See Appendix in Reynolds textbook.)

• A desire to learn!
Course Requirements

Class attendance is recommended
- Outside material will be introduced.

Problem sets
- Problems from textbooks.
- Programming assignments; We will prototype some of our semantics specifications in Coq (or ML/Haskell).

Readings
- Selected chapters in the main textbooks (Harper and Reynolds).
- A couple of research papers.
- Coq (or ML/Haskell) tutorials if you don’t know them.

Grading
- About 70% problem sets, 30% final project/exam (no midterm).
1. Introduction; Predicate Logic
2. Inductive Definitions
3. Abstract Syntax and Binding
4. Imp; Denotational Semantics
5. Failure, Input-Output, and Continuations
6. Static and Dynamic Semantics
7. Program Specifications and Proofs
8. Function Types
9. Plotkin's PCF
10. Finite Data Types
11. Infinite Data Types
12. Untyped Lambda Calculus
13. Dynamic Typing
Syllabus (cont’d)

14. Polymorphic Types
15. Existential Types
16. Control Stacks and Exceptions
17. Continuations
18. Types and Propositions
19. Subtyping; Semantics of Types
20. Storage Effects
21. Monads and Comonads
22. Lazy Evaluation
23. Parallelism
24. Process Calculus
25. Monadic Concurrency
Course Webpage

http://flint.cs.yale.edu/cs430
Predicate Logic
&
Math Background
Predicate Logic

Predicate logic over integer expressions:
  a language of logical assertions, for example

\[ \forall x. x + 0 = x \]

Why discuss predicate logic?

- It is an example of a simple language
- It has simple denotational semantics
- We will use it later in program specifications
Abstract Syntax

Describes the structure of a phrase
ignoring the details of its representation.

An abstract grammar for predicate logic over integer expressions:

\[ \text{intexp} ::= 0 \mid 1 \mid \ldots \]
\[ \mid \text{var} \]
\[ \mid \neg \text{intexp} \mid \text{intexp} + \text{intexp} \mid \text{intexp} - \text{intexp} \mid \ldots \]

\[ \text{assert} ::= \text{true} \mid \text{false} \]
\[ \mid \text{intexp} = \text{intexp} \mid \text{intexp} < \text{intexp} \mid \text{intexp} \leq \text{intexp} \mid \ldots \]
\[ \mid \neg \text{assert} \mid \text{assert} \land \text{assert} \mid \text{assert} \lor \text{assert} \]
\[ \mid \text{assert} \Rightarrow \text{assert} \mid \text{assert} \Leftrightarrow \text{assert} \]
\[ \mid \forall \text{var}. \text{assert} \mid \exists \text{var}. \text{assert} \]
Resolving Notational Ambiguity

- Using parentheses: $(\forall x. (((((x) + (0)) + 0) = (x))))$

- Using precedence and parentheses: $\forall x. (x + 0) + 0 = x$

  arithmetic operators ($\ast / \text{rem} \ldots$) with the usual precedence
  relational operators ($\equiv \neq < \leq \ldots$)

- The body of a quantified term extends to a delimiter.
Carriers and Constructors

- **Carriers:** sets of abstract phrases (e.g. `intexp`, `assert`)
- **Constructors:** specify abstract grammar productions

\[
\begin{align*}
\text{intexp} &::= 0 &\quad \rightarrow &\quad c_0 \in \{\langle \rangle \} \rightarrow \text{intexp} \\
\text{intexp} &::= \text{intexp} + \text{intexp} &\quad \rightarrow &\quad c_+ \in \text{intexp} \times \text{intexp} \rightarrow \text{intexp}
\end{align*}
\]

Note: Independent of the concrete pattern of the production:

\[
\begin{align*}
\text{intexp} &::= \textbf{plus} \text{ intexp intexp} &\quad \rightarrow &\quad c_+ \in \text{intexp} \times \text{intexp} \rightarrow \text{intexp}
\end{align*}
\]

- Constructors must be injective and have disjoint ranges
- Carriers must be either predefined or their elements must be constructible in finitely many constructor applications
Inductive Structure of Carrier Sets

With these properties of constructors and carriers, carriers can be defined inductively:

\[
\begin{align*}
\text{intexp}^{(0)} &= \{\} \\
\text{intexp}^{(j+1)} &= \{c_0\langle\rangle, \ldots\} \cup \{c_+ (x_0, x_1) \mid x_0, x_1 \in \text{intexp}^{(j)}\} \cup \ldots \\
\text{assert}^{(0)} &= \{\} \\
\text{assert}^{(j+1)} &= \{c_{\text{true}}\langle\rangle, c_{\text{false}}\langle\rangle\} \\
&\quad \cup \{c_\text{=} (x_0, x_1) \mid x_0, x_1 \in \text{intexp}^{(j)}\} \cup \ldots \\
&\quad \cup \{c_\neg (x_0) \mid x_0 \in \text{assert}^{(j)}\} \cup \ldots
\end{align*}
\]

\[
\begin{align*}
\text{intexp} &= \bigcup_{j=0}^{\infty} \text{intexp}^{(j)} \\
\text{assert} &= \bigcup_{j=0}^{\infty} \text{assert}^{(j)}
\end{align*}
\]
The meaning of a term \( e \in intexp \) is \( \llbracket e \rrbracket_{intexp} \),
i.e. the function \( \llbracket - \rrbracket_{intexp} \) maps \( intexp \) objects to their meanings.

**What is the set of meanings?**

The meaning \( \llbracket 5 + 37 \rrbracket_{intexp} \) of the term \( 5 + 37 \) could be the integer 42.

(that is, \( c_+ (c_5 \langle \rangle, c_{37} \langle \rangle) \))

However the term \( x + 5 \) contains the free variable \( x \),
so the meaning of an \( intexp \) in general cannot be an integer...
Mathematical Background

- Sets
- Relations
- Functions
- Sequences
- Products and Sums
Sets

\[ x \in S \] membership \quad \{ \} \quad \text{the empty set}
\[ x \notin S \] \quad S = \{ x \} \quad \mathbb{N} \quad \text{natural numbers}
\[ S \subseteq T \] inclusion \quad \mathbb{Z} \quad \text{integers}
\[ S \subseteq^\text{fin} T \] finite subset \quad \mathbb{B} = \{ \text{true, false} \}

\{ E \mid P \} \quad \text{set comprehension}

\[ S \cap T \] intersection \quad = \{ x \mid x \in S \text{ and } x \in T \}
\quad \text{\( x \) is a bound variable}

\[ S \cup T \] union \quad = \{ x \mid x \in S \text{ or } x \in T \}

\[ S - T \] difference \quad = \{ x \mid x \in S \text{ and not } x \in T \}

\[ \mathcal{P} S \] powerset \quad = \{ T \mid T \subseteq S \}

\[ m \text{ to } n \] integer range \quad = \{ x \mid m \leq x \text{ and } x \leq n \}
Generalized Set Operations

\[ \bigcup S \overset{\text{def}}{=} \{ x \mid \exists T \in S. \ x \in T \} \]
\[ \bigcap S \overset{\text{def}}{=} \{ x \mid \forall T \in S. \ x \in T \} \]
\[ \bigcup_{i \in I} S \overset{\text{def}}{=} \bigcup \{ S \mid i \in I \} \]
\[ \bigcap_{i \in I} S \overset{\text{def}}{=} \bigcap \{ S \mid i \in I \} \]
\[ \bigcup_{i=m}^{n} S \overset{\text{def}}{=} \bigcup_{i \in m \text{ to } n} S \]
\[ \bigcap_{i=m}^{n} S \overset{\text{def}}{=} \bigcap_{i \in m \text{ to } n} S \]
\[ \bigcup \{ \} \ = \ \{ \} \]
\[ \bigcap \{ \} \ = \ \text{meaningless} \]

Examples:

\[ A \cup B = \bigcup \{ A, B \} \]
\[ \bigcup \{ i \text{ to } (i + 1) \mid i \in \{ j^2 \mid j \in 1 \text{ to } 3 \} \} = \{ 1, 2, 4, 5, 9, 10 \} \]
Relations

A relation $\rho$ is a set of primitive pairs $[x, y]$.

$\rho$ relates $x$ and $y$ $\iff$ $x \rho y$ $\iff$ $[x, y] \in \rho$

$\rho$ is an identity relation $\iff$ $(\forall x, y. x \rho y \Rightarrow x = y)$

the identity on $S$ $I_S$ $\stackrel{\text{def}}{=} \{ [x, x] \mid x \in S \}$

the domain of $\rho$ $\text{dom} \ \rho$ $\stackrel{\text{def}}{=} \{ x \mid \exists y. x \rho y \}$

the range of $\rho$ $\text{ran} \ \rho$ $\stackrel{\text{def}}{=} \{ x \mid \exists y. y \rho x \}$

composition of $\rho$ with $\rho'$ $\rho' \cdot \rho$ $\stackrel{\text{def}}{=} \{ [x, z] \mid \exists y. x \rho y \text{ and } y \rho' z \}$

reflection of $\rho$ $\rho^\dagger$ $\stackrel{\text{def}}{=} \{ [y, x] \mid [x, y] \in \rho \}$
Relations: Properties and Examples

\((\rho_3 \cdot \rho_2) \cdot \rho_1 = \rho_3 \cdot (\rho_2 \cdot \rho_1)\)

\(\rho \cdot I_S \subseteq \rho \supseteq I_T \cdot \rho\)

\(\text{dom } I_S = S = \text{ran } I_S\)

\(I_T \cdot I_S = I_{T \cap S}\)

\(I_S^{\dagger} = I_S\)

\((\rho^{\dagger})^{\dagger} = \rho\)

\((\rho_2 \cdot \rho_1)^{\dagger} = \rho_1^{\dagger} \cdot \rho_2^{\dagger}\)

\(\rho \cdot \{\} = \{\} = \{\} \cdot \rho\)

\(I_{\{\}} = \{\} = \{\}^{\dagger}\)

\(\text{dom } \rho = \{\} \Rightarrow \rho = \{\}\)

\(I_N = \{[0, 0], [1, 1], [2, 2], \ldots\}\)

\(< = \{[0, 1], [0, 2], [1, 2], \ldots\}\)

\(\leq = \{[0, 0], [0, 1], [1, 1], [0, 2], \ldots\}\)

\(\geq = \{[0, 0], [1, 0], [1, 1], [2, 0], \ldots\}\)

\(< \subseteq \leq\)

\(< \cup I_N = \leq\)

\(\leq \cap \geq = I_N\)

\(< \cap \geq = \{\}\)

\(< \cdot \leq = <\)

\(\leq \cdot \leq = \leq\)

\(\geq = \leq^{\dagger}\)
Functions

A relation \( f \) is a **function** if

\[
\forall x, x', x''. \ ( [x, x'] \in f \text{ and } [x, x''] \in f ) \implies x' = x'
\]

If \( f \) is a function,

\[
f x = y \iff f_x = y \iff f \text{ maps } x \text{ to } y \iff [x, y] \in f
\]

\( I_S \) and \( \{\} \) are functions.

If \( f \) and \( g \) are functions, then \( g \cdot f \) is a function: \((g \cdot f) x = g(f x)\)

\( f^\dagger \) is not necessarily a function:

consider \( f = \{ [\text{true}, \{\}], [\text{false}, \{\}] \} \)

\( f \) is an **injection** if both \( f \) and \( f^\dagger \) are functions.
Notation for Functions

**Typed abstraction:** \( \lambda x \in S. \ E \overset{\text{def}}{=} \{ [x, E] \mid x \in S \} \)

Defined only when \( E \) is defined for all \( x \in S \)
(consider \( \lambda g \in \mathbb{N}. \ g \ 3 \))

\( I_S = \lambda x \in S. \ x \)

\( g \cdot f = \lambda x \in \text{dom} \ f. \ g(f \ x), \) if \( \text{ran} \ f \subseteq \text{dom} \ g. \)

**Placeholder:** \( E \) with a dash \((-)\) standing for the bound variable

\( g(-) \ h = \lambda x \in S. \ (g(x)) \ h \quad - + 42 = \lambda x \in \mathbb{N}. \ x + 42 \)

**Variation** of a function \( f \): \( [f \mid x : y] \ z = \)

\[
\begin{cases} 
  y, & \text{if } z = x \\
  f \ z, & \text{otherwise}
\end{cases}
\]

\( \text{dom} \ [f \mid x : y] = (\text{dom} \ f) \cup \{x\} \)

\( \text{ran} \ [f \mid x : y] = (\text{ran} \ f) - \{z \mid [x, z] \in f\} \cup \{y\} \)
Sequences

\[
\begin{align*}
[f \mid x_1 : y_1 \mid \ldots \mid x_n : y_n] \overset{\text{def}}{=} & \ldots [f \mid x_1 : y_1] \ldots \mid x_n : y_n \\
[x_1 : y_1 \mid \ldots \mid x_n : y_n] \overset{\text{def}}{=} & \{\} \mid x_1 : y_1 \mid \ldots \mid x_n : y_n \\
\langle x_0, \ldots x_{n-1} \rangle \overset{\text{def}}{=} & [0 : x_0 \mid \ldots n-1 : x_{n-1}]
\end{align*}
\]

\[
[\;] = \{\} \; — \; \text{the empty function}
\]

\[
\langle \rangle = [\;] = \{\} \; — \; \text{the empty sequence}
\]

\[
\langle x_0, \ldots x_{n-1} \rangle \; — \; \text{an } n\text{-tuple}
\]

\[
\langle x, y \rangle \; — \; \text{a (non-primitive) pair}
\]

\[
\text{dom} \langle x_0, \ldots x_{n-1} \rangle = 0 \text{ to } (n-1)
\]

\[
\langle x_0, \ldots x_{n-1} \rangle_i = x_i \; \text{when } i \in 0 \text{ to } (n-1)
\]
Products

Let \( \theta \) be an indexed family of sets (a function with sets in its range). The \textbf{Cartesian product} of \( \theta \) is

\[
\prod \theta \overset{\text{def}}{=} \{ f \mid \text{dom } f = \text{dom } \theta \text{ and } \forall i \in \text{dom } \theta. f(i) \in \theta(i) \}
\]

\[
\prod \langle B, B \rangle
\]

\[
= \prod (\lambda x \in 0 \text{ to } 1. B)
\]

\[
= \{ [0 : \text{true}, 1 : \text{true}], [0 : \text{true}, 1 : \text{false}], [0 : \text{false}, 1 : \text{true}], [0 : \text{false}, 1 : \text{false}] \}
\]

\[
= \{ \langle \text{true, true} \rangle, \langle \text{true, false} \rangle, \langle \text{false, true} \rangle, \langle \text{false, false} \rangle \}
\]
More Products

\[ \prod_{x \in T} S \overset{\text{def}}{=} \prod \lambda x \in T. S \]
\[ \prod_{i=m}^{n} S \overset{\text{def}}{=} \prod_{i \in (m \text{ to } n)} S \]
\[ S_1 \times \ldots \times S_n \overset{\text{def}}{=} \prod_{i=1}^{n} "S_i" \]
\[ S^T \overset{\text{def}}{=} \prod_{x \in T} S \]
\[ S^n \overset{\text{def}}{=} S^0 \text{ to } (n-1) = \underbrace{S \times \ldots \times S}_{n \text{ times}} \]

\[ \prod \langle B, B \rangle = B \times B = B^2 \]
\[ S^0 = S^{\{\}} = \{\{\}\} = \{\{\}\} \]
Sets of Sequences

\[
S^+ \overset{\text{def}}{=} \bigcup_{i=1}^{\infty} S^i \\
S^* \overset{\text{def}}{=} S^0 \cup S^+ \\
S^\infty \overset{\text{def}}{=} S^* \cup S^\mathbb{N}
\]

Let \( U = \{\langle \rangle\} \)

\[
U^+ = \{\langle \rangle, \langle \langle \rangle \rangle, \langle \langle \langle \rangle \rangle \rangle, \langle \langle \langle \langle \rangle \rangle \rangle \rangle, \ldots \} \quad \text{(finite)}
\]

\[
U^* = \{\langle \rangle, \langle \langle \rangle \rangle, \langle \langle \langle \rangle \rangle \rangle, \langle \langle \langle \langle \rangle \rangle \rangle \rangle, \ldots \} \quad \text{(finite)}
\]

\[
U^\infty = \{\langle \rangle, \langle \langle \rangle \rangle, \langle \langle \langle \rangle \rangle \rangle, \langle \langle \langle \langle \rangle \rangle \rangle \rangle, \ldots \} \quad \text{(infinite)}
\]
Sums

Let $\theta$ be an indexed family of sets (a function with sets in its range). The disjoint union (sum) of $\theta$ is

$$\sum \theta \overset{\text{def}}{=} \{ \langle i, x \rangle \mid i \in \text{dom} \theta \text{ and } x \in \theta i \}$$

$$\sum_{x \in T} S \overset{\text{def}}{=} \sum \lambda x \in T. S \quad S_1 + \ldots + S_n \overset{\text{def}}{=} \sum_{i=1}^{n} \ldots \overset{\text{def}}{=}$

$$\sum_{i=m}^{n} S \overset{\text{def}}{=} \sum_{i \in (m \text{ to } n)} S \quad T \times S = \sum_{x \in T} S$$

$$n \times S = (0 \text{ to } (n - 1)) \times S = \underbrace{S + \ldots + S}_{n \text{ times}}$$

$$\mathbf{B} + \mathbf{B} = \sum \langle \mathbf{B}, \mathbf{B} \rangle = \{ \langle 0, \text{true} \rangle, \langle 0, \text{false} \rangle, \langle 1, \text{true} \rangle, \langle 1, \text{false} \rangle \}$$

$$= 2 \times \mathbf{B}$$
Functions of Multiple Arguments

- Use **tuples** instead of multiple arguments:

\[
\begin{align*}
  f(a_0, \ldots a_{n-1}) & \longrightarrow f \langle a_0, \ldots a_{n-1} \rangle \\
\end{align*}
\]

Syntactic sugar:

\[
\begin{align*}
  \lambda \langle x_0 \in S_0, \ldots, x_{n-1} \in S_{n-1} \rangle. E & \equiv \\
  \lambda x \in S_0 \times \ldots \times S_{n-1}. (\lambda x_0 \in S_0. \ldots \lambda x_{n-1} \in S_{n-1}. E) \\
  (x_0) \ldots (x(n-1)) \\
\end{align*}
\]

- Use **Currying**:

\[
\begin{align*}
  f(a_0, \ldots a_{n-1}) & \longrightarrow f a_0 \ldots a_{n-1} \\
  \equiv (\ldots (f a_0) \ldots) a_{n-1} \\
\end{align*}
\]

where \( f \) is a Curried function \( \lambda x_0 \in S_0. \ldots \lambda x_{n-1} \in S_{n-1}. E \).
Relations Between Sets

ρ is a relation from S to T

\[ \iff \rho \in S \xrightarrow{\text{REL}} T \]
\[ \iff \text{dom } \rho \subseteq S \text{ and ran } \rho \subseteq T. \]

Relation on S  \( \overset{\text{def}}{=} \) relation from S to S.

\[ I_S \in S \xrightarrow{\text{REL}} S \]
\[ \rho \in S \xrightarrow{\text{REL}} T \implies \rho^\dagger \in T \xrightarrow{\text{REL}} S \]

For all S and T, \( \emptyset \in S \xrightarrow{\text{REL}} T \)
\[ \emptyset \in! S \xrightarrow{\text{REL}} \emptyset \]
\[ \emptyset \in! \emptyset \xrightarrow{\text{REL}} T \]
Total Relations

\( \rho \in S_{\text{REL}} \rightarrow T \) is a total relation from \( S \) to \( T \)

\( \iff \rho \in S_{\text{TREL}} \rightarrow T \)

\( \iff \forall x \in S. \exists y \in T. x \rho y \)

\( \iff \text{dom } \rho = S \)

\( \iff I_S \subseteq \rho^\dagger \cdot \rho \)

\( \rho \in (\text{dom } \rho)_{\text{TREL}} \rightarrow T \iff T \supseteq \text{ran } \rho \)
Functions Between Sets

\( f \) is a partial function from \( S \) to \( T \)

\[ \iff f \in S \xrightarrow{\text{PFUN}} T \]

\[ \iff f \in S \xrightarrow{\text{REL}} T \text{ and } f \text{ is a function.} \]

“Partial”: \( f \in S \xrightarrow{\text{REL}} T \Rightarrow \text{dom } f \subseteq S \)

\( f \in S \xrightarrow{\text{PFUN}} T \) is a (total) function from \( S \) to \( T \)

\[ \iff f \in S \rightarrow T \]

\[ \iff \text{dom } f = S. \]

- \( S \rightarrow T = T^S = \prod_{x \in S} T \)
- \( S \rightarrow T \rightarrow U = S \rightarrow (T \rightarrow U) \)
Surjections, Injections, Bijections

\[ f \text{ is a surjection from } S \text{ to } T \iff \text{ran } f = T \]

\[ f \text{ is a injection from } S \text{ to } T \iff f^\dagger \in T \xrightarrow{\text{PFUN}} S \]

\[ f \text{ is a bijection from } S \text{ to } T \iff f^\dagger \in T \rightarrow S \]

\[ \iff f \text{ is an isomorphism from } S \text{ to } T \]
Back to Predicate Logic

\[ \text{intexp} ::= 0 \mid 1 \mid \ldots \]
\[ \quad \mid \text{var} \]
\[ \quad \mid \neg \text{intexp} \mid \text{intexp} + \text{intexp} \mid \text{intexp} - \text{intexp} \mid \ldots \]
\[ \text{assert} ::= \text{true} \mid \text{false} \]
\[ \quad \mid \text{intexp} = \text{intexp} \mid \text{intexp} < \text{intexp} \mid \text{intexp} \leq \text{intexp} \mid \ldots \]
\[ \quad \mid \neg \text{assert} \mid \text{assert} \land \text{assert} \mid \text{assert} \lor \text{assert} \]
\[ \quad \mid \text{assert} \Rightarrow \text{assert} \mid \text{assert} \Leftrightarrow \text{assert} \]
\[ \quad \mid \forall \text{var}. \text{assert} \mid \exists \text{var}. \text{assert} \]
The meaning of term $e \in \text{intexp}$ is $\llbracket e \rrbracket_{\text{intexp}}$
i.e. the function $\llbracket - \rrbracket_{\text{intexp}}$ maps objects from $\text{intexp}$ to their meanings.

What is the set of meanings?

The meaning $\llbracket 5 + 37 \rrbracket_{\text{intexp}}$ of the term $5 + 37$ could be the integer 42.

But the term $x + 5$ contains the free variable $x$...
Environments

...hence we need an environment (variable assignment, state)

\[\sigma \in \Sigma \overset{\text{def}}{=} \text{var} \rightarrow \mathbb{Z}\]

to give meaning to free variables.

The meaning of a term is a function from the states to \( \mathbb{Z} \) or \( \mathbb{B} \).

\[
\begin{align*}
\llbracket \_ \rrbracket_{\text{intexp}} & \in \text{intexp} \rightarrow \Sigma \rightarrow \mathbb{Z} \\
\llbracket \_ \rrbracket_{\text{assert}} & \in \text{assert} \rightarrow \Sigma \rightarrow \mathbb{B}
\end{align*}
\]

if \( \sigma = [x: 3, y: 4] \), then \( \llbracket x+5 \rrbracket_{\text{intexp}} \sigma = 8 \)

\( \llbracket \exists z. x < z \land z < y \rrbracket \sigma = \text{false} \)
Direct Semantics Equations for Predicate Logic

\[ v \in \text{var} \quad e \in \text{intexp} \quad p \in \text{assert} \]

\[
\begin{align*}
\llbracket 0 \rrbracket \text{intexp} \sigma &= 0 \\
(\text{really } \llbracket c_0 \langle \rangle \rrbracket \text{intexp} \sigma &= 0) \\
\llbracket v \rrbracket \text{intexp} \sigma &= \sigma v \\
\llbracket e_0 + e_1 \rrbracket \text{intexp} \sigma &= \llbracket e_0 \rrbracket \text{intexp} \sigma + \llbracket e_1 \rrbracket \text{intexp} \sigma \\
\llbracket \text{true} \rrbracket \text{assert} \sigma &= \text{true} \\
\llbracket e_0 = e_1 \rrbracket \text{assert} \sigma &= \llbracket e_0 \rrbracket \text{intexp} \sigma = \llbracket e_1 \rrbracket \text{intexp} \sigma \\
\llbracket \neg p \rrbracket \text{assert} \sigma &= \neg (\llbracket p \rrbracket \text{assert} \sigma) \\
\llbracket p_0 \land p_1 \rrbracket \text{assert} \sigma &= \llbracket p_0 \rrbracket \text{assert} \sigma \land \llbracket p_1 \rrbracket \text{assert} \sigma \\
\llbracket \forall v. p \rrbracket \text{assert} \sigma &= \forall n \in \mathbb{Z}. \llbracket p \rrbracket \text{assert} [\sigma | v : n] 
\end{align*}
\]
Example: The Meaning of a Term

\[
[\forall x. x+0=x]_{assert}\sigma
\]

\[
= \forall n \in \mathbb{Z}. [x+0=x]_{assert}[\sigma|x : n]
\]

\[
= \forall n \in \mathbb{Z}. [x+0]_{intexp}[\sigma|x : n] = [x]_{intexp}[\sigma|x : n]
\]

\[
= \forall n \in \mathbb{Z}. [x]_{intexp}[\sigma|x : n] + [0]_{intexp}[\sigma|x : n] = [x]_{intexp}[\sigma|x : n]
\]

\[
= \forall n \in \mathbb{Z}. [\sigma|x : n](x) + 0 = [\sigma|x : n](x)
\]

\[
= \forall n \in \mathbb{Z}. n + 0 = n
\]

\[
= \text{true}
\]
Properties of the Semantic Equations

- They are **syntax-directed (homomorphomic)**:
  - exactly one equation for each abstract grammar production (constructor)
  - result expressed using functions (meanings) of subterms only (arguments of constructor)

  \[ \Rightarrow \text{they have exactly one solution } \langle \llbracket - \rrbracket_{\text{intexp}}, \llbracket - \rrbracket_{\text{assert}} \rangle \]
  (proof by induction on the structure of terms).

- They define **compositional** semantic functions
  (depending only on the meaning of the subterms)

  \[ \Rightarrow \text{“equivalent” subterms can be substituted} \]
Validity of Assertions

$p$ holds/is true in $\sigma$ $\iff$ $\sigma$ satisfies $p$ $\iff$ $[p]_{\text{assert}} \sigma = \text{true}$

$p$ is valid $\iff$ $\forall \sigma \in \Sigma. p$ holds in $\sigma$

$p$ is unsatisfiable $\iff$ $\forall \sigma \in \Sigma. [p]_{\text{assert}} \sigma = \text{false}$ $\iff$ $\neg p$ is valid

$p$ is stronger than $p'$ $\iff$ $\forall \sigma \in \Sigma. (p'$ holds if $p$ holds $)$ $\iff$ $(p \Rightarrow p')$ is valid

$p$ and $p'$ are equivalent $\iff$ $p$ is stronger than $p'$ $\text{and } p'$ is stronger than $p$
## Inference Rules

<table>
<thead>
<tr>
<th>Class</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ ( p ) (Axiom)</td>
<td>⊢ ( x + 0 = x ) (xPlusZero)</td>
</tr>
<tr>
<td>⊢ ( p ) (Axiom Schema)</td>
<td>( ⊢ e_1 = e_0 \Rightarrow e_0 = e_1 ) (SymmObjEq)</td>
</tr>
<tr>
<td>( ⊢ p_0 ) ( \ldots ) ( ⊢ p_{n-1} ) (Rule)</td>
<td>( ⊢ p ) ( ⊢ p \Rightarrow p' ) (ModusPonens)</td>
</tr>
<tr>
<td>( ⊢ p ) ( ⊢ p' ) (ModusPonens)</td>
<td>( ⊢ \forall v. p ) (Generalization)</td>
</tr>
</tbody>
</table>
A set of inference rules defines a **logical theory** $\vdash$.

A **formal proof** (in a logical theory):

a sequence of **instances** of the inference rules, where the premisses of each rule occur as conclusions earlier in the sequence.

1. $\vdash x + 0 = x$ (xPlusZero)
2. $\vdash x + 0 = x \Rightarrow x = x + 0$ (SymmObjEq) $[e_0 : x | e_1 : x + 0 ]$
3. $\vdash x = x + 0$ (ModusPonens, 1, 2) $[p : x + 0 = x | p' : x = x + 0 ]$
4. $\vdash \forall x. x = x + 0$ (Generalization, 3) $[v : x | p : x = x + 0 ]$
Tree Representation of Formal Proofs

\[
\begin{align*}
\vdash x + 0 &= x & \vdash x + 0 &= x \Rightarrow x = x + 0 & \text{(SymmObjEq)} \\
\vdash x = x + 0 & \vdash \forall x. x = x + 0 & \text{(Gen)} \\
\end{align*}
\]
An inference rule is sound if in every instance of the rule the conclusion is valid if all the premisses are.

A logical theory \( \vdash \) is sound if all inference rules in it are sound.

If \( \vdash \) is sound and there is a formal proof of \( \vdash p \), then \( p \) is valid.

Object vs Meta implication:

\[ \vdash p \Rightarrow \forall v. p \text{ is not a sound rule, although } \frac{\vdash p}{\vdash \forall v. p} \text{ is.} \]
Completeness of a Logical Theory

A logical theory $\vdash$ is **complete** if

for every valid $p$ there is a formal proof of $\vdash p$.

A logical theory $\vdash$ is **axiomatizable** if

there exists a finite set of inference rules

from which can be constructed formal proofs of all assertions in $\vdash$.

No first-order theory of arithmetic is complete and axiomatizable.
Variable Binding

∀x. ∃y. x < y ∧ ∃x. x > y
Variable Binding

∀x. ∃y. x < y ∧ ∃x. x > y
Bound and Free Variables

In $\forall v. \, p$, $v$ is the **binding occurrence** (**binder**) and $p$ is its **scope**.

If a non-binding occurrence of $v$ is within the scope of a binder for $v$, then it is a **bound** occurrence; otherwise it’s a **free** one.

\[
\begin{align*}
FV_{intexp}(0) & = \{\} & FV_{assert}(\text{true}) & = \{\} \\
FV(v) & = \{v\} & FV(e_0=\text{e}_1) & = FV(e_0) \cup FV(e_1) \\
FV(-e) & = FV(e) & FV(\neg p) & = FV(p) \\
FV(e_0 + \text{e}_1) & = FV(e_0) \cup FV(e_1) & FV(p_0 \land p_1) & = FV(p_0) \cup FV(p_1) \\
FV(\forall v. \, p) & = FV(p) - \{v\}
\end{align*}
\]

Example:

\[
FV(\exists y. \, x < y \land \exists x. \, x > y) = \{x\}
\]
Only Assignment of Free Variables Matters

Coincidence Theorem:
If $\sigma v = \sigma' v$ for all $v \in FV_\theta(p)$, then $[p]_\theta \sigma = [p]_\theta \sigma'$
(where $p$ is a phrase of type $\theta$).

Proof: By structural induction.

Inductive hypothesis:
The statement of the theorem holds for all phrases of depth less than that of the phrase $p'$.

Base cases:
$p' = 0 \Rightarrow [0]_{intexp} \sigma = 0 = [0]_{intexp} \sigma'$
$p' = v \Rightarrow [v]_{intexp} \sigma = \sigma v = \sigma' v = [v]_{intexp} \sigma'$, since $FV(v) = \{v\}$. 
Proof of Coincidence Theorem, cont’d

Coincidence Theorem:

If $\sigma v = \sigma' v$ for all $v \in FV_\theta(p)$, then $[p]_\theta \sigma = [p]_\theta \sigma'$.

Inductive cases:

$p' = e_0 + e_1$: by IH $[e_i]_{\text{intexp}} \sigma = [e_i]_{\text{intexp}} \sigma'$, $i \in \{1, 2\}$.

$[p']_{\text{intexp}} \sigma = [e_0]_{\text{intexp}} \sigma + [e_1]_{\text{intexp}} \sigma$

$= [e_0]_{\text{intexp}} \sigma' + [e_1]_{\text{intexp}} \sigma' = [p']_{\text{intexp}} \sigma'$

$p' = \forall u. q$: $\sigma v = \sigma' v$, $\forall v \in FV(p') = FV(q) - \{u\}$

then $[\sigma | u : n] v = [\sigma' | u : n] v$, $\forall v \in FV(q)$, $n \in \mathbb{Z}$

Then by IH $[q]_{\text{assert}} [\sigma | u : n] = [q]_{\text{assert}} [\sigma' | u : n]$ for all $n \in \mathbb{Z}$,

hence $\forall n \in \mathbb{Z}. [q]_{\text{assert}} [\sigma | u : n] = \forall n \in \mathbb{Z}. [q]_{\text{assert}} [\sigma' | u : n]$

$[\forall u. q]_{\text{assert}} \sigma = [\forall u. q]_{\text{assert}} \sigma'$. 
Substitution

\[
\begin{align*}
-\delta / \delta & \in \text{intexp} \rightarrow \text{intexp} \\
-\delta / \delta & \in \text{assert} \rightarrow \text{assert}
\end{align*}
\] when \(\delta \in \text{var} \rightarrow \text{intexp}\)

\[
\begin{align*}
0 / \delta &= 0 \\
(-e) / \delta &= -(e / \delta) \\
(e_0 + e_1) / \delta &= (e_0 / \delta) + (e_1 / \delta) \\
\ldots \\
v / \delta &= \delta v \\
(p_0 \land p_1) / \delta &= (p_0 / \delta) \land (p_1 / \delta) \\
(\forall v. p) / \delta &= \forall v'. (p / [\delta | v : v']), \\
\text{where } v' &\not\in \bigcup_{u \in \text{FV}(p) - \{v\}} \text{FV}(\delta u)
\end{align*}
\]

Examples:

\[
\begin{align*}
(x < 0 \land \exists x. x \leq y) / [x : y+1] &= y + 1 < 0 \land \exists x. x \leq y \\
(x < 0 \land \exists x. x \leq y) / [y : x+1] &= x < 0 \land \exists z. z \leq x + 1
\end{align*}
\]
Preserving Binding Structure

\[(x < 0 \land \exists x. x \leq y)/[x : y+1] = y+1 < 0 \land \exists x. x \leq y\]
Avoiding Variable Capture

\[(x < 0 \land \exists x. x \leq y)/[\text{\textcolor{blue}{y}} : x+1] = x < 0 \land \exists z. z \leq x+1\]
Substitution Theorems

Substitution Theorem:
If $\sigma = \llbracket - \rrbracket_{intexp}\sigma' \cdot \delta$ on $FV(p)$, then $\llbracket - \rrbracket \sigma p = (\llbracket - \rrbracket \sigma' \cdot (-/\delta)) p$.

Finite Substitution Theorem:
$\llbracket p/v_0 \rightarrow e_0, \ldots v_{n-1} \rightarrow e_{n-1} \rrbracket \sigma = \llbracket p \rrbracket [\sigma | v_0 : \llbracket e_0 \rrbracket \sigma, \ldots ]$.

where
$p/v_0 \rightarrow e_0, \ldots v_{n-1} \rightarrow e_{n-1} \overset{\text{def}}{=} p/[c\text{var}| v_0 : e_0 | \ldots | v_{n-1} : e_{n-1}]$.

Renaming:
If $u \notin FV(q) - \{v\}$, then $\llbracket \forall u. (q/v \rightarrow u) \rrbracket_{boolexp} = \llbracket \forall v. q \rrbracket_{boolexp}$. 