CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Course Overview

January 14, 2025

Today’s Lecture

- Why study formal semantics?
- How I 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.

USS Yorktown

"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.

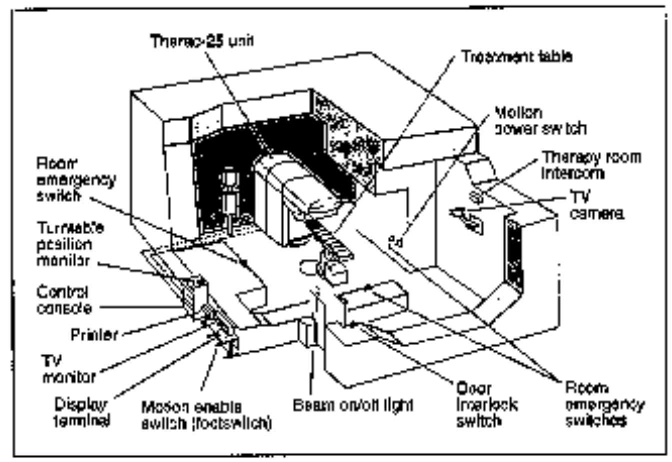


Figure 1. Typical Therac-25 facility.

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?

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.

Certified Heterogeneous Systems

- How to build efficient, scalable, and trustworthy heterogeneous systems?

  Need a high-level architectural design + stepwise refinement

- Correct-by-Construction or Secure-by-Construction
  - HW/SW Implementation → Deep/Fully-Abstract Functional Spec

    (VeriLog, C, Asm) (written in some formal logic)

    (semantics for these languages) (need formal proof assistant)
  - Mechanized proofs for the above "implements" relation

- Need a theory of component composition
  - What is a component? (HW vs. SW ones)
  - What is a "certified" component?
  - What are different ways of connecting/composing these components?

Sample Research Themes

- Shared-memory concurrency & concurrent objects
- Virtual memory management & spatial isolation
- File and storage systems and device drivers
- OS kernel and hypervisor for heterogeneous architecture
- Secure enclaves
- Web server
- Blockchains and smart contracts
- Consensus-based distributed systems
- Efficient proof-certificate checking

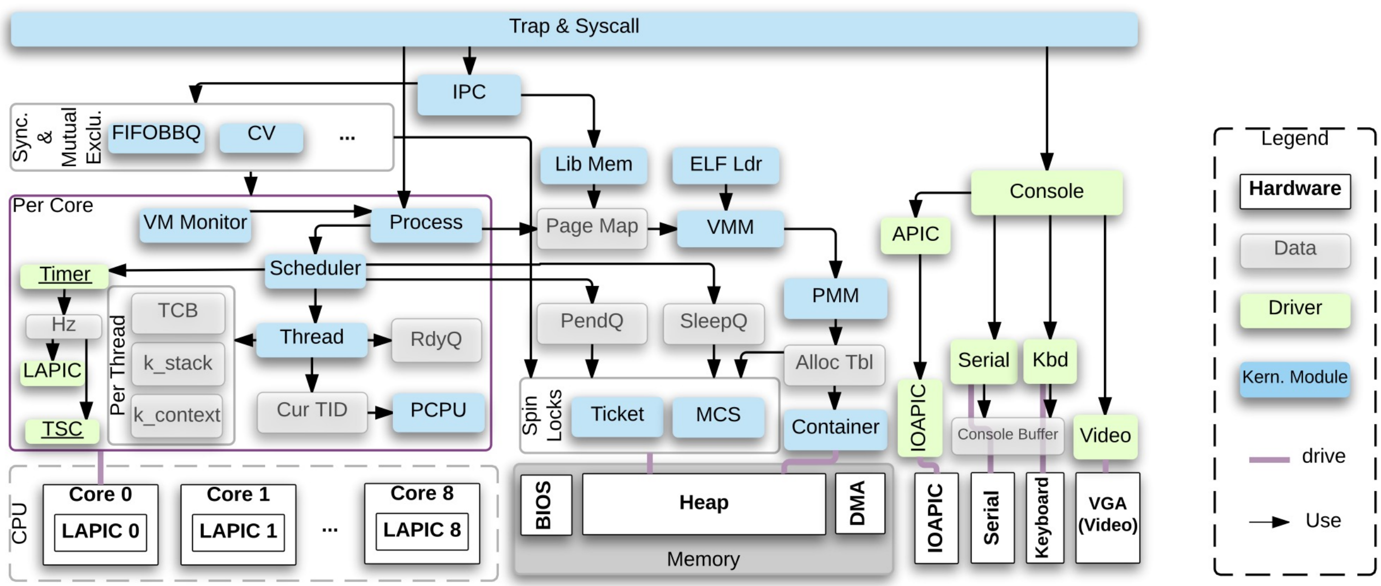
Problem Definition

- What is a certified OS kernel / hypervisor / security monitor?
  - a system binary ***implements*** its specification running over a HW machine model (w. devices & interrupts)?
  - what should the specification & the machine model be like?

- What properties do we want to prove?
  - safety & partial correctness properties
  - total functional correctness
  - security properties (isolation, confidentiality, integrity, availability)
  - resource usage properties (stack overflow, real time properties)
  - race-freedom, atomicity, and linearizability
  - liveness properties (deadlock-freedom, starvation freedom)

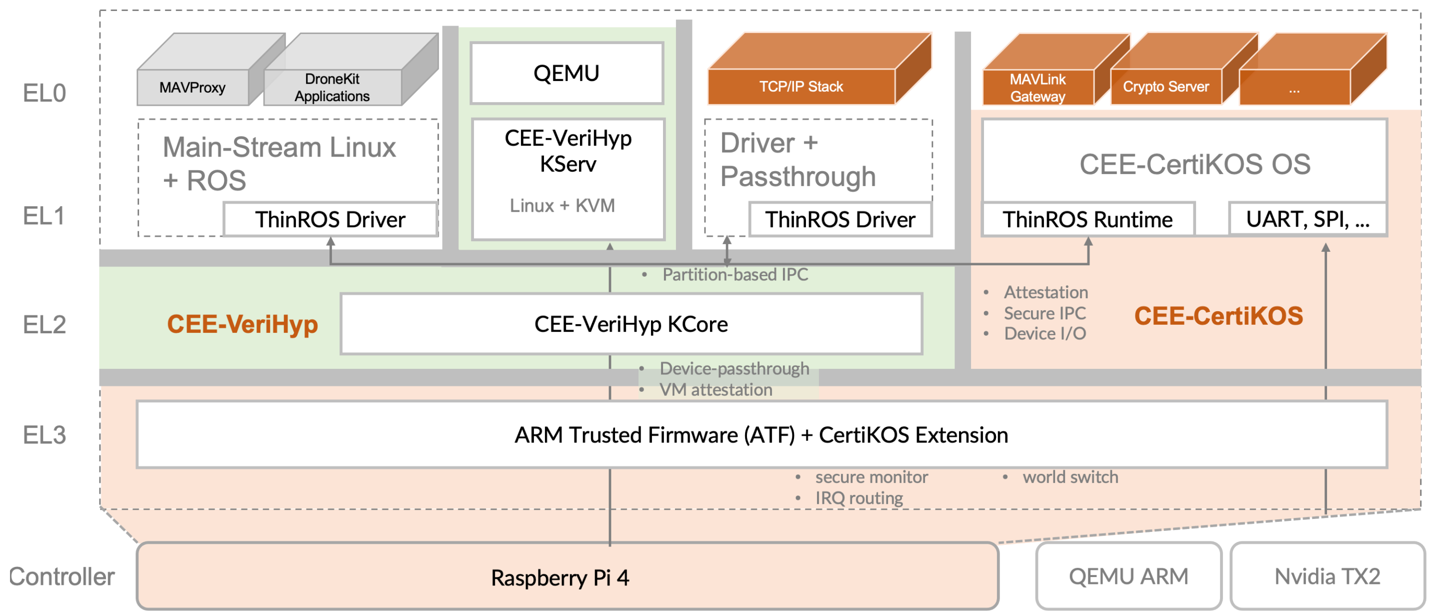
- How to cut down the cost of verification?

Problem Definition: Example OS Kernel



Formally Verified Concurrent CertiKOS (mC2) [OSDI 2016]

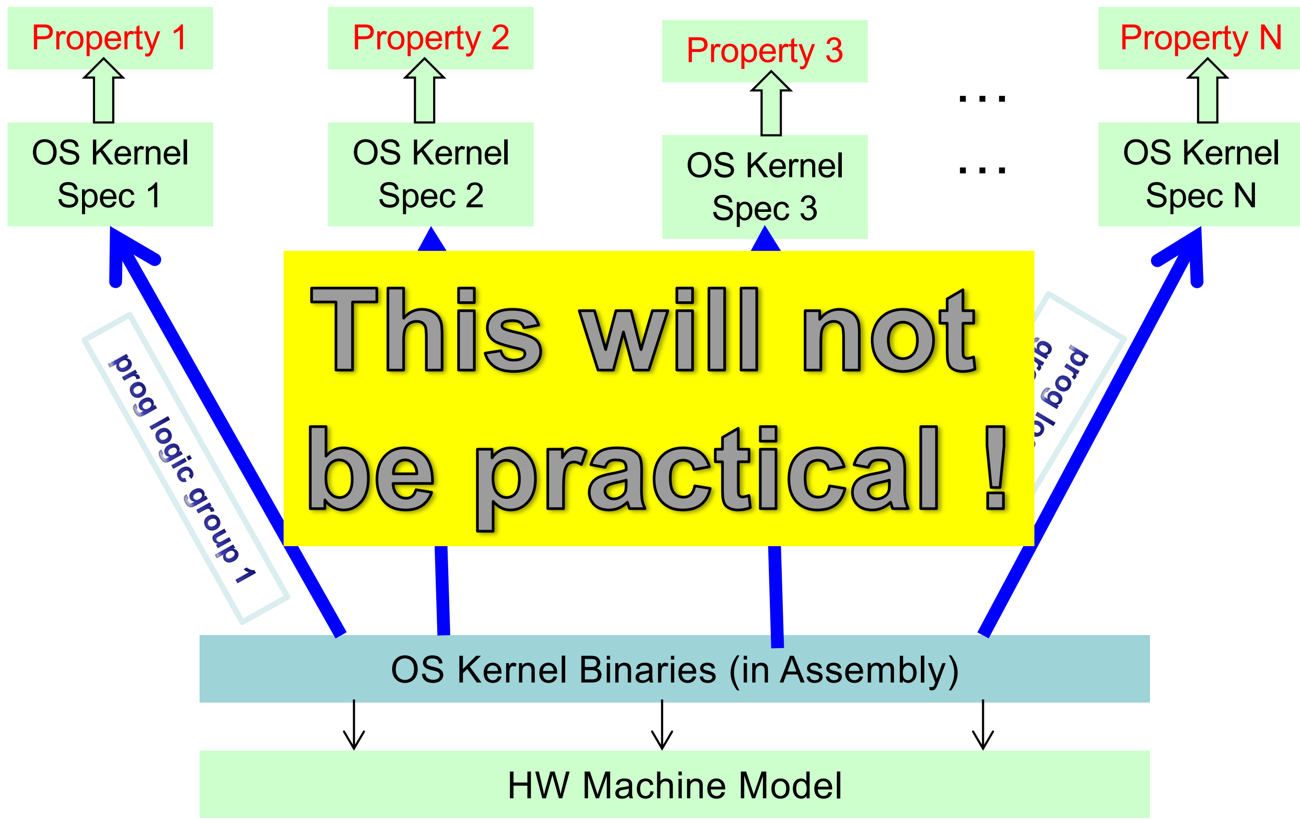
Problem Definition: Example Deployment



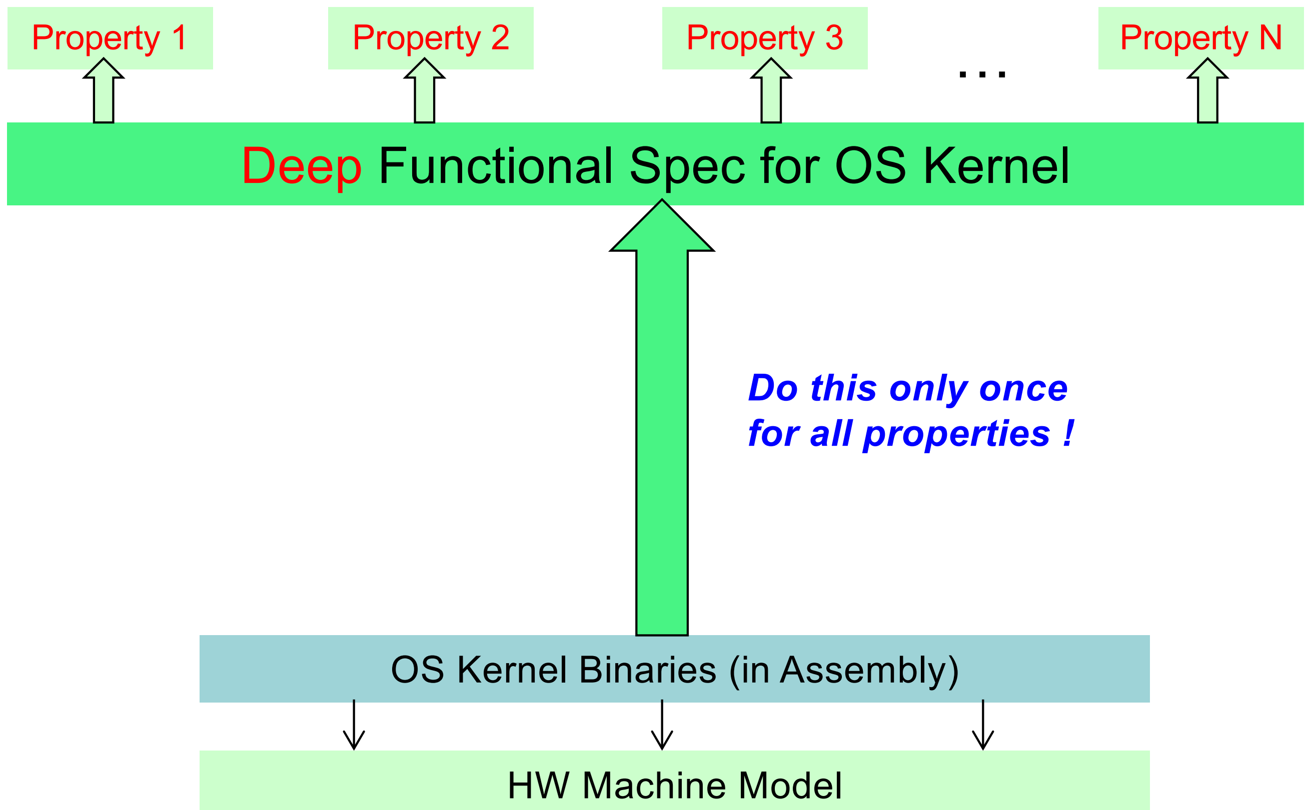
REFUEL: Formally Verified Composition of Secure Enclaves

[Joint w. Columbia U., DARPA V-SPELLS 2021-2025]

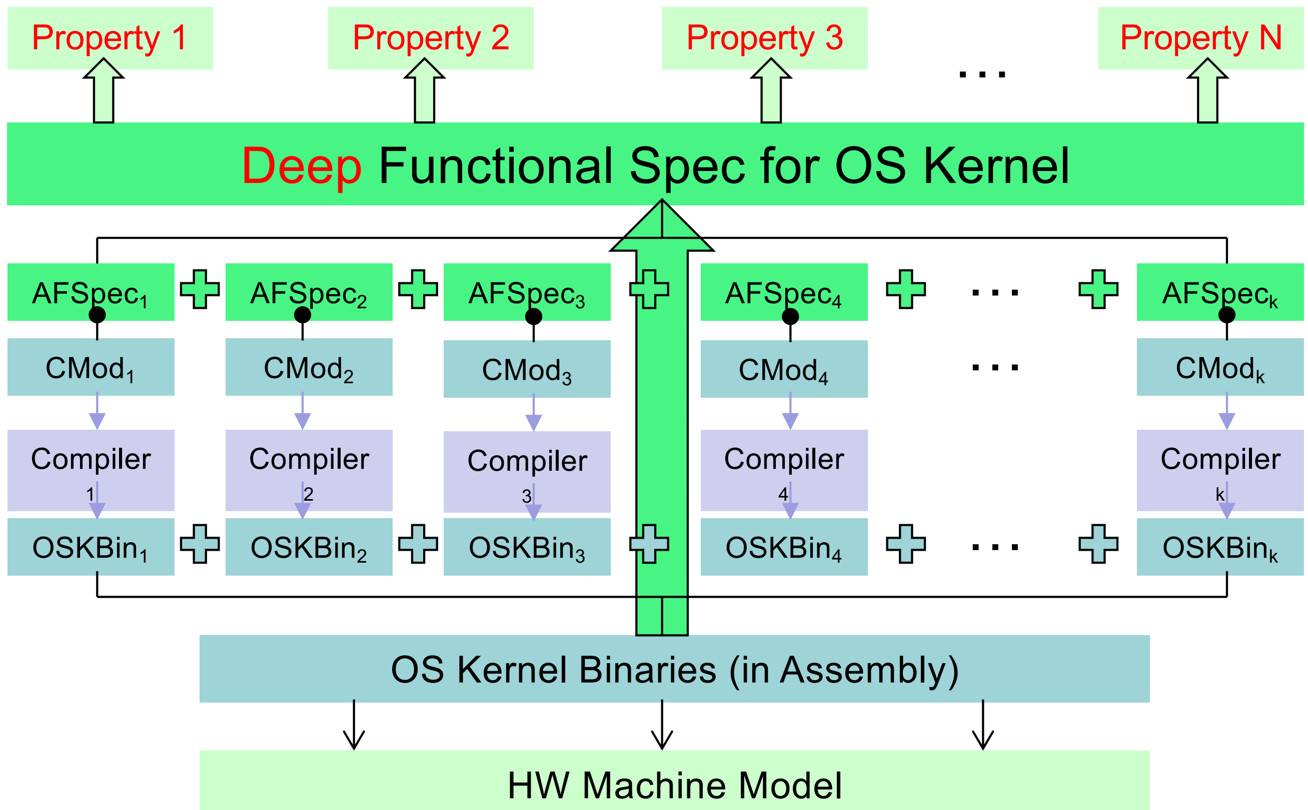
OS Verification: The Conventional Approach



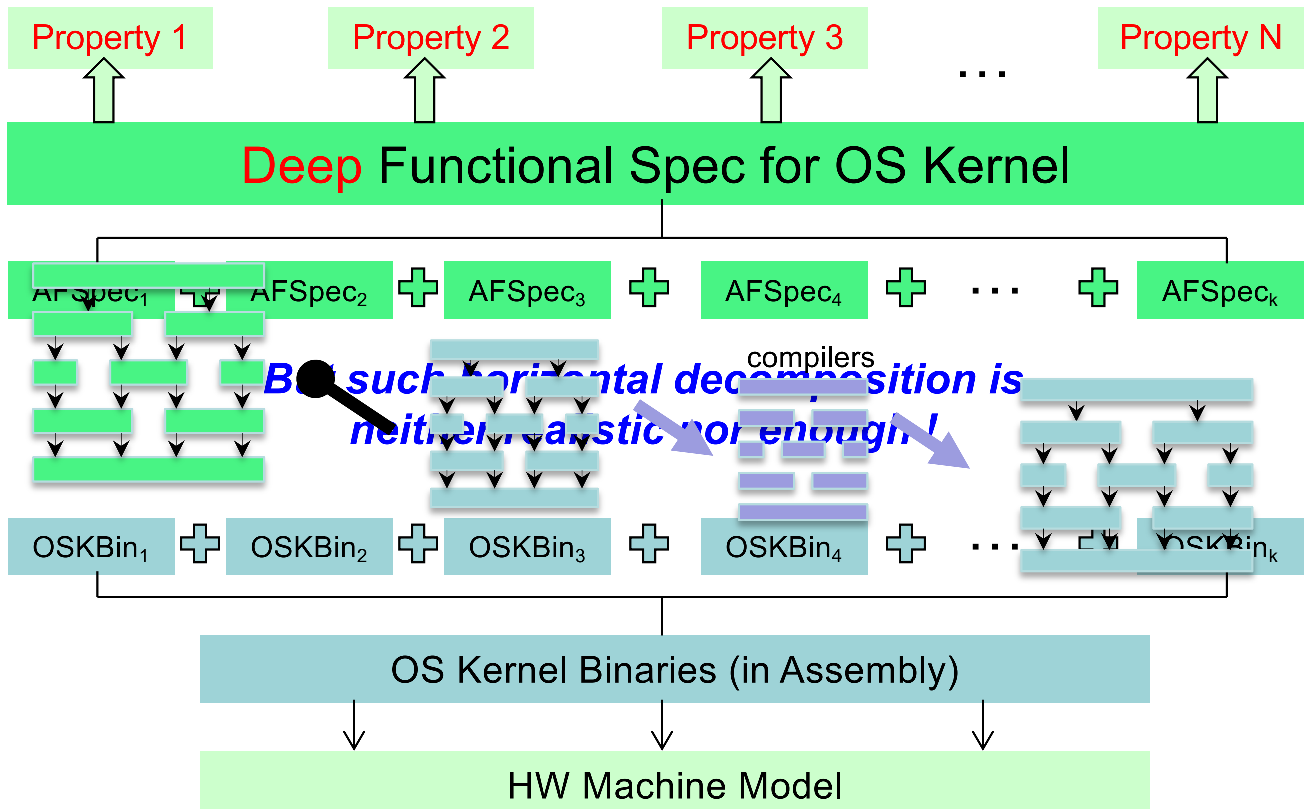
The CertiKOS Approach



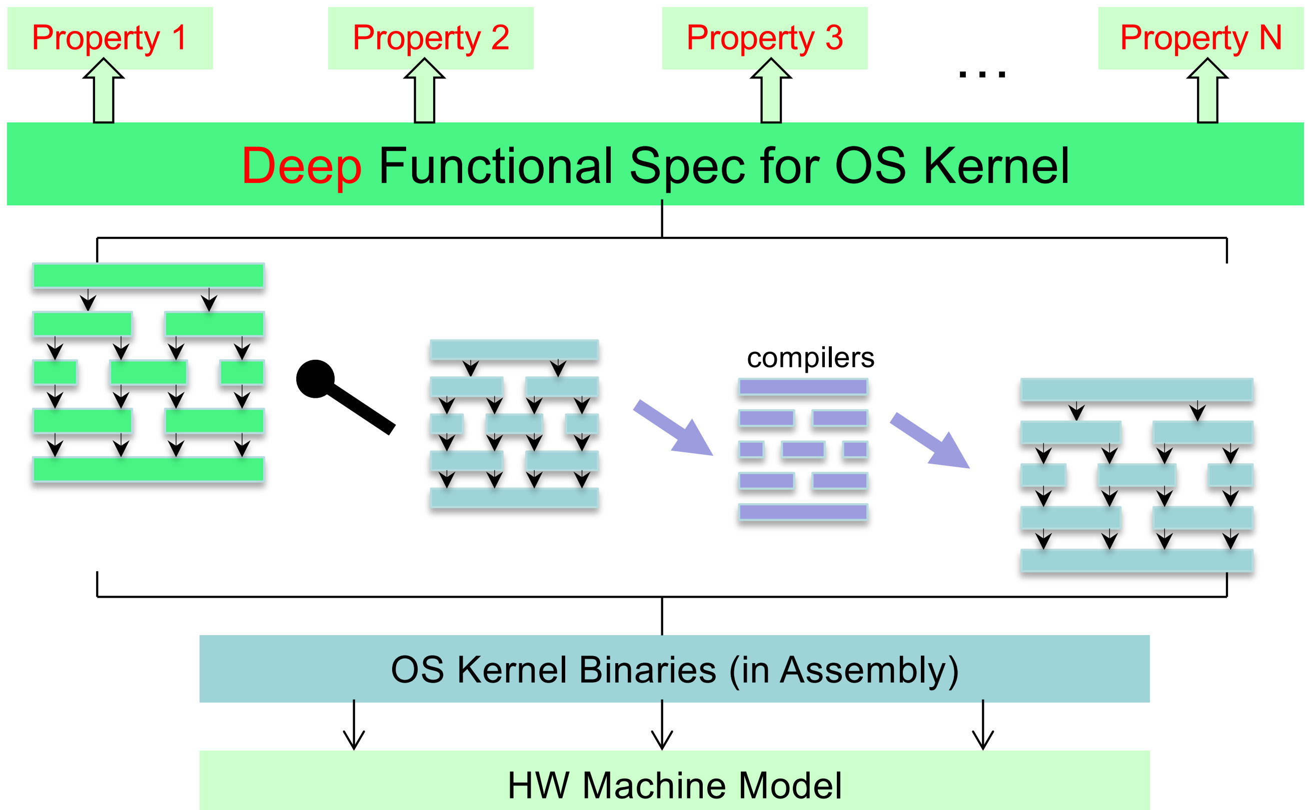
The CertiKOS Approach



The CertiKOS Approach



The CertiKOS Approach



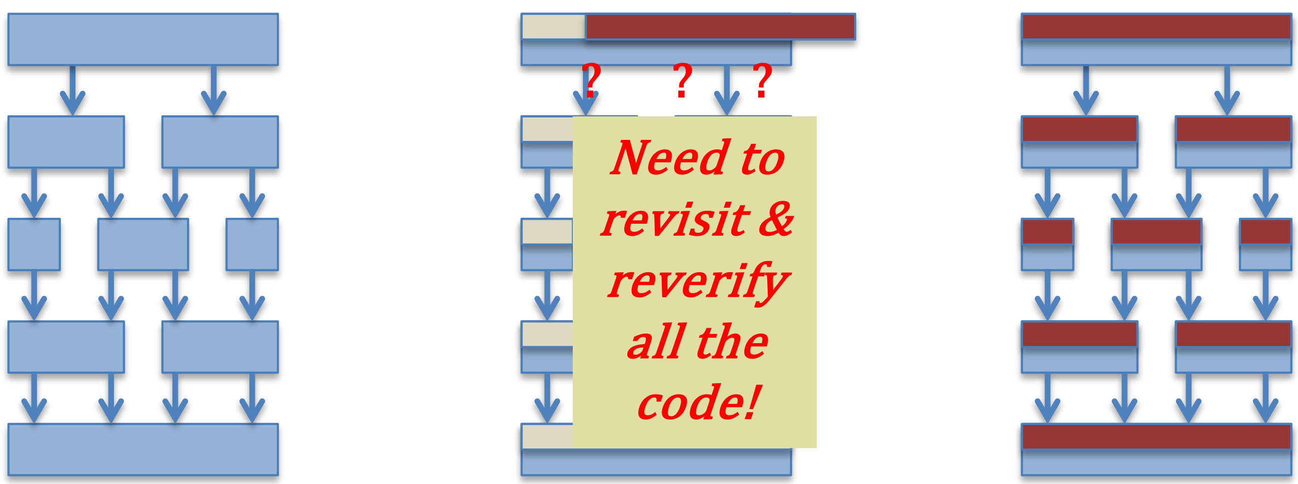
What is a Deep Spec?

C or Asm module | rich spec A | rich spec B

C & Asm Module Implementation

C & Asm Modules w. rich spec A

Want to prove another spec B ?



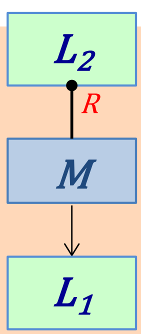
What is a Deep Spec?

$[\![M]\!] L_1 \sim_R L_2$

$[\![M]\!] (L_1)$ and L_2 simulates each other!

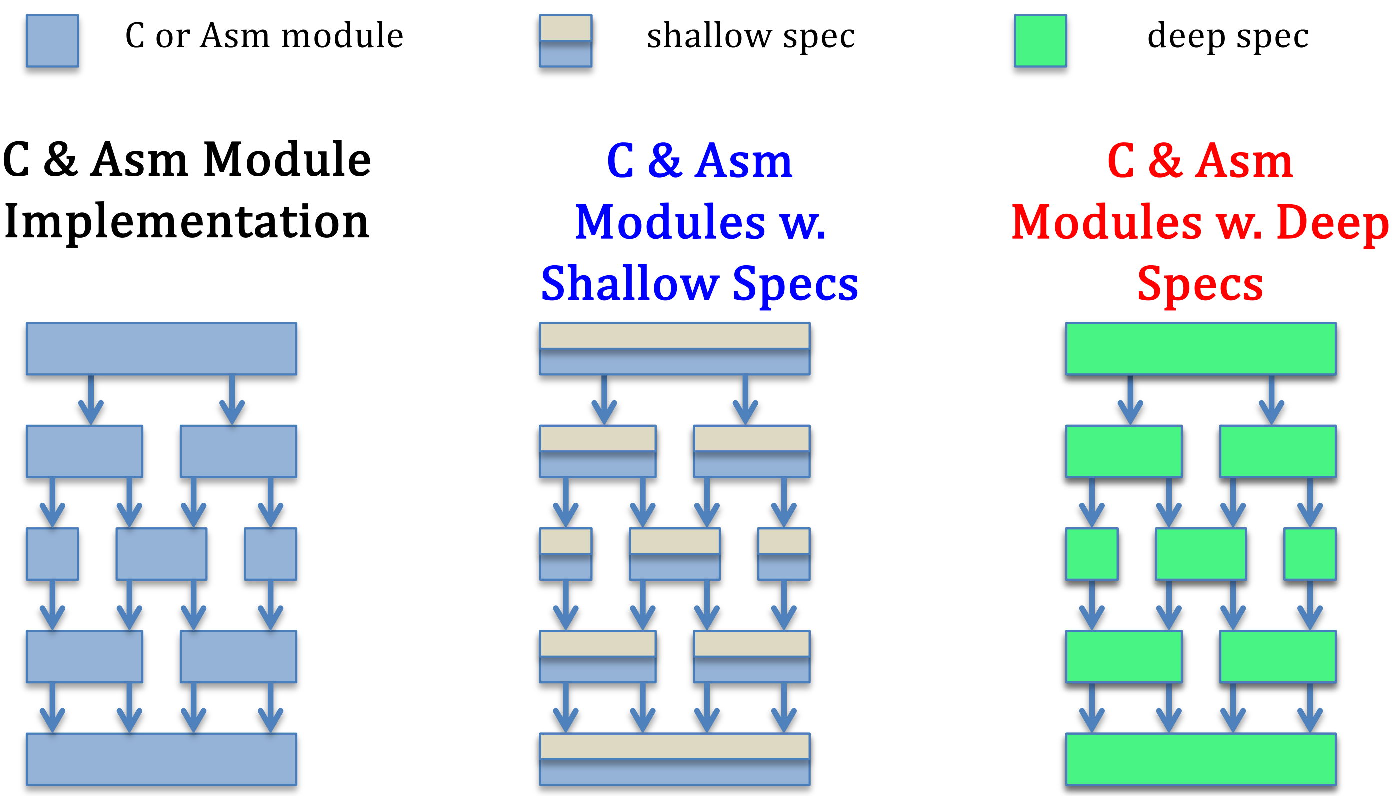
L_2 captures everything about running M over L_1

Making it “contextual” using
the whole-program semantics 【•】



L_2 is a **deep specification** of M over L_1
if under any valid program context P of L_2,
【 $P \oplus M$ 】 (L_1) and 【 P 】 (L_2) are
observationally equivalent

Shallow vs. Deep Specifications



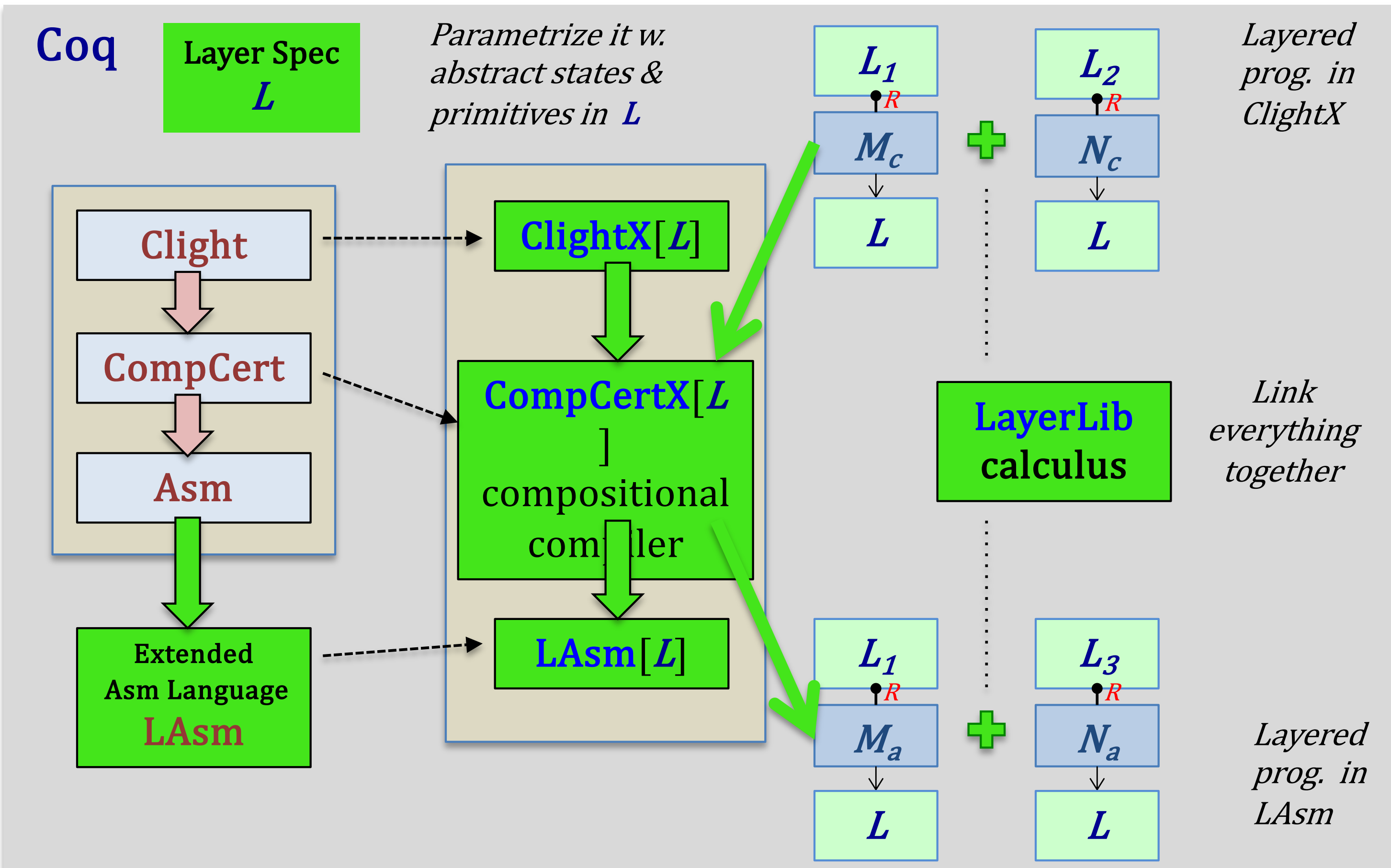
The CertiKOS Approach

- We developed a language-based formalization of certified abstraction layers with deep specifications

- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - LAsm for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into LAsm layers

- We built multiple certified OS kernels in Coq
  - The initial version has 37 layers and can boot Linux as a guest
  - The later versions support interrupts & multicore concurrency & security (spatial & temporal isolation w. real-time guarantee)

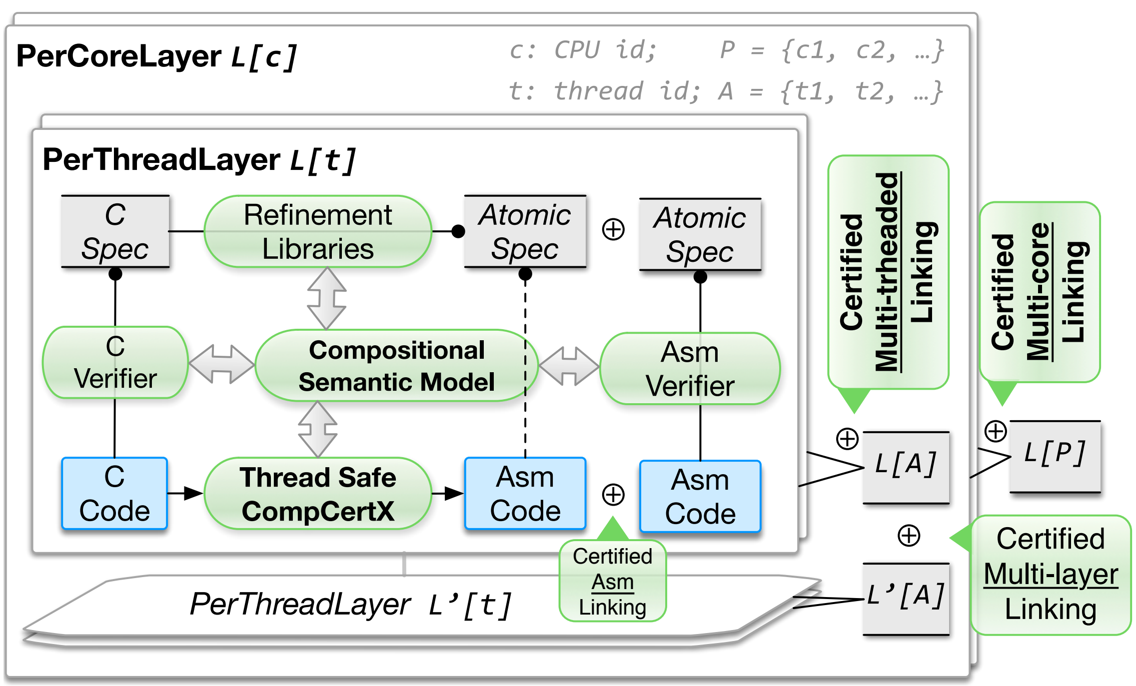
The CertiKOS Toolchain (CAL) [POPL'15]



The CertiKOS Toolchain (CCAL) [PLDI'18]

New programming toolkit w. certified multicore & multithreaded linking:

Composition = parallel composition + hiding (abstraction)



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.

Readings

- Selected chapters in the main textbooks (Harper and Reynolds).
- A couple of research papers.
- Coq tutorials if you don't know them.

Grading

- About 75% problem sets, 25% final project.

Syllabus

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

$$\forall \mathsf{x}.\ \mathsf{x} + 0 = \mathsf{x}$$

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:

$$
\begin{aligned}
\mathit{intexp} ::= {} & \mathsf{0} \mid \mathsf{1} \mid \ldots \\
& \mid \mathit{var} \\
& \mid -\mathit{intexp} \mid \mathit{intexp} + \mathit{intexp} \mid \mathit{intexp} - \mathit{intexp} \mid \ldots
\end{aligned}
$$

$$
\begin{aligned}
\mathit{assert} ::= {} & \mathbf{true} \mid \mathbf{false} \\
& \mid \mathit{intexp} = \mathit{intexp} \mid \mathit{intexp} < \mathit{intexp} \mid \mathit{intexp} \leq \mathit{intexp} \mid \ldots \\
& \mid \neg \mathit{assert} \mid \mathit{assert} \wedge \mathit{assert} \mid \mathit{assert} \vee \mathit{assert} \\
& \mid \mathit{assert} \Rightarrow \mathit{assert} \mid \mathit{assert} \Leftrightarrow \mathit{assert} \\
& \mid \forall \mathit{var}.\ \mathit{assert} \mid \exists \mathit{var}.\ \mathit{assert}
\end{aligned}
$$

Resolving Notational Ambiguity

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

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

  arithmetic operators ($*$ / **rem** ...) with the usual precedence
  relational operators ($= \neq < \leq\ \ldots$)
  $\neg$
  $\wedge$
  $\vee$
  $\Rightarrow$
  $\Leftrightarrow$

- 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

$$intexp ::= 0 \longrightarrow c_0 \in \{\langle\rangle\} \to intexp$$
$$intexp ::= intexp + intexp \longrightarrow c_+ \in intexp \times intexp \to intexp$$

Note: Independent of the concrete pattern of the production:

$$intexp ::= \mathbf{plus}\ intexp\ intexp \longrightarrow c_+ \in intexp \times intexp \to intexp$$

- 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:

$$
\begin{aligned}
intexp^{(0)} &= \{\} \\
intexp^{(j+1)} &= \{c_0\langle\rangle, \ldots\} \cup \{c_+(x_0, x_1) \mid x_0, x_1 \in intexp^{(j)}\} \cup \ldots \\
assert^{(0)} &= \{\} \\
assert^{(j+1)} &= \{c_{\mathbf{true}}\langle\rangle,\ c_{\mathbf{false}}\langle\rangle\} \\
&\quad \cup \{c_=(x_0, x_1) \mid x_0, x_1 \in intexp^{(j)}\} \cup \ldots \\
&\quad \cup \{c_\neg(x_0) \mid x_0 \in assert^{(j)}\} \cup \ldots
\end{aligned}
$$

$$
\begin{aligned}
intexp &= \bigcup_{j=0}^{\infty} intexp^{(j)} \\
assert &= \bigcup_{j=0}^{\infty} assert^{(j)}
\end{aligned}
$$

Denotational Semantics of Predicate Logic

The meaning of a term $e \in \mathit{intexp}$ is $[\![e]\!]_{\mathit{intexp}}$

i.e. the function $[\![-]\!]_{\mathit{intexp}}$ maps intexp objects to their meanings.

What is the set of meanings?

The meaning $[\![\mathsf{5} + \mathsf{37}]\!]_{\mathit{intexp}}$ of the term $\underbrace{\mathsf{5} + \mathsf{37}}_{\text{(that is, } c_{+}(c_{5}\langle\rangle, c_{37}\langle\rangle))}$ could be the integer 42.

However the term $\mathsf{x} + \mathsf{5}$ 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	$\{\}$	the empty set
$x \in! S$	$S = \{x\}$	$\mathbf{N}$	natural numbers
$S \subseteq T$	inclusion	$\mathbf{Z}$	integers
$S \subseteq^{\mathsf{fin}} T$	finite subset	$\mathbf{B}$	$= \{\mathbf{true}, \mathbf{false}\}$

$\{E \mid P\}$	set comprehension	
$S \cap T$	intersection	$= \{x \mid x \in S \text{ and } x \in T\}$ x is a bound variable
$S \cup T$	union	$= \{x \mid x \in S \text{ or } x \in T\}$
$S - T$	difference	$= \{x \mid x \in S \text{ and not } x \in T\}$
$\mathcal{P}\, S$	powerset	$= \{T \mid T \subseteq S\}$
$m \ \mathbf{to}\ n$	integer range	$= \{x \mid m \leq x \text{ and } x \leq n\}$

Generalized Set Operations

$\cup S \stackrel{\text{def}}{=} \{x \mid \exists T \in S.\, x \in T\}$	$\cap S \stackrel{\text{def}}{=} \{x \mid \forall T \in S.\, x \in T\}$
$\bigcup_{i \in I} S \stackrel{\text{def}}{=} \cup\{S \mid i \in I\}$	$\bigcap_{i \in I} S \stackrel{\text{def}}{=} \cap\{S \mid i \in I\} \ldots$
$\bigcup_{i=m}^{n} S \stackrel{\text{def}}{=} \bigcup_{i \in\, m \,\textbf{to}\, n} S$	$\bigcap_{i=m}^{n} S \stackrel{\text{def}}{=} \bigcap_{i \in\, m \,\textbf{to}\, n} S$
$\cup\{\} = \{\}$	$\cap\{\}$ meaningless

Examples:

$$A \cup B = \cup\{A, B\}$$

$$\cup\left\{i \,\textbf{to}\, (i+1) \;\middle|\; i \in \{j^2 \mid j \in 1 \,\textbf{to}\, 3\}\right\} = \{1, 2, 4, 5, 9, 10\}$$

Relations

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

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

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

the identity on S	I_S	$\stackrel{\text{def}}{=}$	$\{[x, x] \mid x \in S\}$
the domain of ρ	$\text{dom}\, \rho$	$\stackrel{\text{def}}{=}$	$\{x \mid \exists y.\, x \,\rho\, y\}$
the range of ρ	$\text{ran}\, \rho$	$\stackrel{\text{def}}{=}$	$\{x \mid \exists y.\, y \,\rho\, x\}$
composition of ρ with ρ'	$\rho' \cdot \rho$	$\stackrel{\text{def}}{=}$	$\{[x, z] \mid \exists y.\, x \,\rho\, y \text{ and } y \,\rho'\, z\}$
reflection of ρ	$\rho^\dagger$	$\stackrel{\text{def}}{=}$	$\{[y, x] \mid [x, y] \in \rho\}$

Relations: Properties and Examples

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

$$\rho \cdot I_S \subseteq \rho \supseteq I_T \cdot \rho$$

$$\mathrm{dom}\, I_S = S = \mathrm{ran}\, I_S$$

$$I_T \cdot I_S = I_{T \cap S}$$

$${I_S}^\dagger = I_S$$

$$(\rho^\dagger)^\dagger = \rho$$

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

$$\rho \cdot \{\} = \{\} = \{\} \cdot \rho$$

$$I_{\{\}} = \{\} = \{\}^\dagger$$

$$\mathrm{dom}\, \rho = \{\} \Rightarrow \rho = \{\}$$

$$I_{\mathbf{N}} = \{[0,0],[1,1],[2,2],\ldots\}$$

$$< \; = \{[0,1],[0,2],[1,2],\ldots\}$$

$$\leq \; = \{[0,0],[0,1],[1,1],[0,2],\ldots\}$$

$$\geq \; = \{[0,0],[1,0],[1,1],[2,0],\ldots\}$$

$$< \; \subseteq \; \leq$$

$$< \cup I_{\mathbf{N}} = \; \leq$$

$$\leq \cap \geq \; = I_{\mathbf{N}}$$

$$< \cap \geq \; = \{\}$$

$$< \cdot \leq \; = \; <$$

$$\leq \cdot \leq \; = \; \leq$$

$$\geq \; = \; \leq^\dagger$$

Functions

A relation f is a function if

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

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 $f = \{[\mathbf{true}, \{\}], [\mathbf{false}, \{\}]\}$

f is an injection if both f and $f^\dagger$ are functions.

Notation for Functions

Typed abstraction: $\lambda x \in S.\, E \stackrel{\text{def}}{=} \{[x, E] \mid x \in S\}$

Defined only when E is defined for all $x \in S$

(consider $\lambda g \in \mathbf{N}.\, g\ 3$)

$I_S = \lambda x \in S.\, x$

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

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

$g\,(-)\,h \;=\; \lambda x \in S.\,(g\,(x))\,h \qquad -+42 \;=\; \lambda x \in \mathbf{N}.\, x+42$

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

$$[f \mid x_1 : y_1 \mid \ldots \mid x_n : y_n] \stackrel{\text{def}}{=} [\ldots[f \mid x_1 : y_1] \ldots \mid x_n : y_n]$$

$$[x_1 : y_1 \mid \ldots \mid x_n : y_n] \stackrel{\text{def}}{=} [\{\} \mid x_1 : y_1 \mid \ldots \mid x_n : y_n]$$

$$\langle x_0, \ldots x_{n-1} \rangle \stackrel{\text{def}}{=} [0 : x_0 \mid \ldots n-1 : x_{n-1}]$$

$[\,] = \{\}$ — the empty function

$\langle\rangle = [\,] = \{\}$ — the empty sequence

$\langle x_0, \ldots x_{n-1} \rangle$ — an n-tuple

$\langle x, y \rangle$ — a (non-primitive) pair

dom $\langle x_0, \ldots x_{n-1} \rangle = 0$ **to** $(n-1)$

$\langle x_0, \ldots x_{n-1} \rangle_i = x_i$ when $i \in 0$ **to** $(n-1)$

Products

Let θ be an indexed family of sets (a function with sets in its range).

The Cartesian product of θ is

$$\prod \theta \stackrel{\text{def}}{=} \{f \mid \text{dom } f = \text{dom } \theta \text{ and } \forall i \in \text{dom } \theta.\, f\ i \in \theta\ i\}$$

$$\begin{aligned}
&\prod \langle \mathbf{B}, \mathbf{B} \rangle \\
&\quad = \prod(\lambda x \in 0 \text{ to } 1.\, \mathbf{B}) \\
&\quad = \{[0 : \mathbf{true}, 1 : \mathbf{true}],\ [0 : \mathbf{true}, 1 : \mathbf{false}], \\
&\qquad [0 : \mathbf{false}, 1 : \mathbf{true}],\ [0 : \mathbf{false}, 1 : \mathbf{false}]\} \\
&\quad = \{\langle \mathbf{true}, \mathbf{true} \rangle,\ \langle \mathbf{true}, \mathbf{false} \rangle,\ \langle \mathbf{false}, \mathbf{true} \rangle,\ \langle \mathbf{false}, \mathbf{false} \rangle\}
\end{aligned}$$

More Products

$$\prod_{x \in T} S \stackrel{\text{def}}{=} \prod \lambda x \in T.\, S$$

$$S_1 \times \ldots \times S_n \stackrel{\text{def}}{=} \prod_{i=1}^{n} \text{“}S_i\text{”}$$

$$\prod_{i=m}^{n} S \stackrel{\text{def}}{=} \prod_{i \in (m \textbf{ to } n)} S$$

$$S^T \stackrel{\text{def}}{=} \prod_{x \in T} S$$

$$S^n \stackrel{\text{def}}{=} S^{0 \textbf{ to } (n-1)} = \underbrace{S \times \ldots \times S}_{n \text{ times}}$$

$$\prod \langle \mathbf{B}, \mathbf{B} \rangle = \mathbf{B} \times \mathbf{B} = \mathbf{B}^2$$

$$S^0 = S^{\{\}} = \{\langle\rangle\} = \{\{\}\}$$

Sets of Sequences

Let $\mathbf{U} = \{\langle\rangle\}$

$$S^{+} \stackrel{\text{def}}{=} \bigcup_{i=1}^{\infty} S^{i} \qquad \mathbf{U}^{+} = \{\langle\langle\rangle\rangle,\ \langle\langle\rangle,\langle\rangle\rangle,\langle\langle\rangle,\langle\rangle,\langle\rangle\rangle,\ldots\text{(finite)}\}$$

$$S^{*} \stackrel{\text{def}}{=} S^{0} \cup S^{+} \qquad \mathbf{U}^{*} = \{\langle\rangle,\langle\langle\rangle\rangle,\ \langle\langle\rangle,\langle\rangle\rangle,\langle\langle\rangle,\langle\rangle,\langle\rangle\rangle,\ldots\text{(finite)}\}$$

$$S^{\infty} \stackrel{\text{def}}{=} S^{*} \cup S^{\mathbf{N}} \qquad \mathbf{U}^{\infty} = \{\langle\rangle,\langle\langle\rangle\rangle,\ \langle\langle\rangle,\langle\rangle\rangle,\langle\langle\rangle,\langle\rangle,\langle\rangle\rangle,\ldots\text{(infinite)}\}$$

Sums

Let θ be an indexed family of sets (a function with sets in its range).

The disjoint union (sum) of θ is

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

$$\sum_{x \in T} S \stackrel{\text{def}}{=} \textstyle\sum \lambda x \in T.\, S \qquad\qquad S_1 + \ldots + S_n \stackrel{\text{def}}{=} \displaystyle\sum_{i=1}^{n} \text{“}S_i\text{”}$$

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

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

$$\begin{aligned}\mathbf{B} + \mathbf{B} = \textstyle\sum\langle \mathbf{B}, \mathbf{B}\rangle &= \{\langle 0, \mathbf{true}\rangle, \langle 0, \mathbf{false}\rangle, \langle 1, \mathbf{true}\rangle, \langle 1, \mathbf{false}\rangle\} \\ &= 2 \times \mathbf{B}\end{aligned}$$

Functions of Multiple Arguments

- Use tuples instead of multiple arguments:

$$f\ (a_0,\ \ldots a_{n-1}) \quad \longrightarrow \quad f\ \langle a_0,\ \ldots a_{n-1}\rangle$$

Syntactic sugar:

$$\lambda\langle x_0 \in S_0,\ \ldots,\ x_{n-1} \in S_{n-1}\rangle.\ E$$

$$\stackrel{\text{def}}{=}\ \lambda x \in S_0 \times \ldots \times S_{n-1}.\ (\lambda x_0 \in S_0.\ \ldots \lambda x_{n-1} \in S_{n-1}.\ E)$$
$$(x\ 0) \ldots (x(n-1))$$

- Use Currying:

$$f\ (a_0,\ \ldots a_{n-1}) \quad \longrightarrow \quad f\ a_0\ \ldots\ a_{n-1}$$
$$= (\ldots (f\ a_0)\ \ldots)\ a_{n-1}$$

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

$$\iff \rho \in S \xrightarrow[\text{REL}]{} T$$

$$\iff \text{dom}\,\rho \subseteq S \text{ and } \text{ran}\,\rho \subseteq T.$$

Relation on S $\stackrel{\text{def}}{=}$ relation from S to S.

$$I_S \in S \xrightarrow[\text{REL}]{} S$$

$$\rho \in S \xrightarrow[\text{REL}]{} T \;\Rightarrow\; \rho^{\dagger} \in T \xrightarrow[\text{REL}]{} S$$

For all S and T, $\{\} \in S \xrightarrow[\text{REL}]{} T$

$$\{\} \in! \; S \xrightarrow[\text{REL}]{} \{\}$$

$$\{\} \in! \; \{\} \xrightarrow[\text{REL}]{} T$$

Total Relations

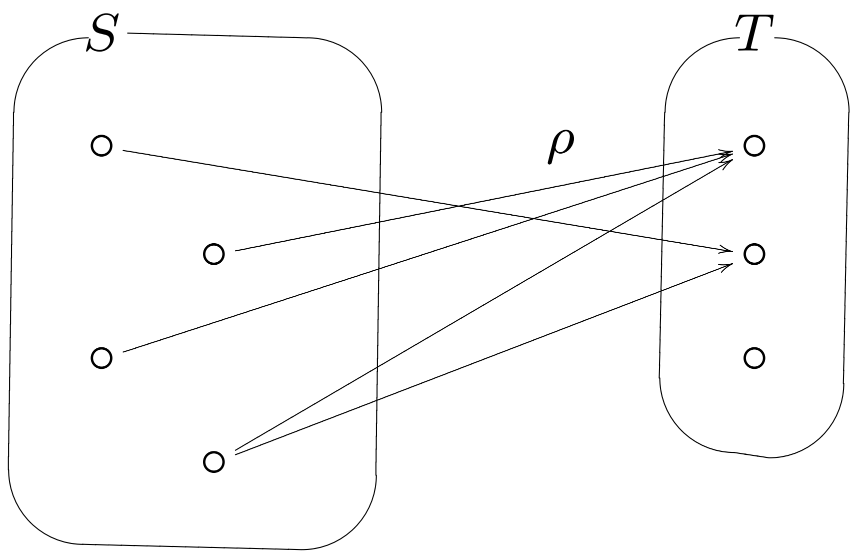
$\rho \in S \xrightarrow[\text{REL}]{} T$ is a total relation from S to T

$$\iff \rho \in S \xrightarrow[\text{TREL}]{} T$$

$$\iff \forall x \in S.\, \exists y \in T.\, x \; \rho \; y$$

$$\iff \text{dom } \rho = S$$

$$\iff I_S \subseteq \rho^{\dagger} \cdot \rho$$



$$\rho \in (\text{dom } \rho) \xrightarrow[\text{TREL}]{} T \iff T \supseteq \text{ran } \rho$$

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

$$\iff f \in S \to T$$

$$\iff \text{dom } f = S.$$

- $S \to T = T^S = \prod_{x \in S} T$
- $S \to T \to U = S \to (T \to U)$

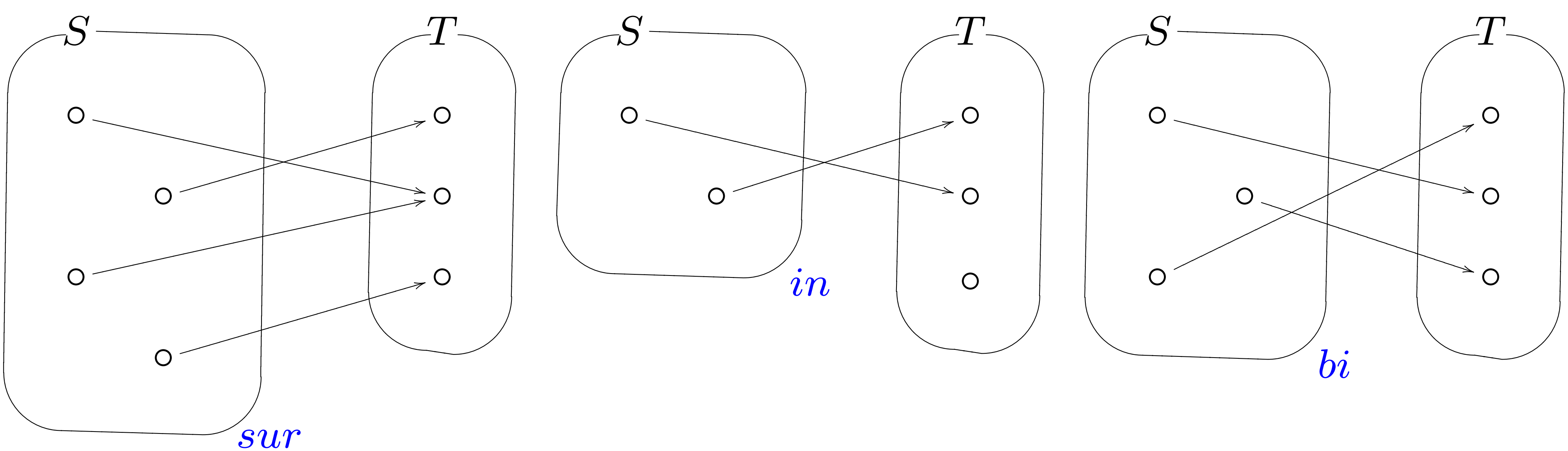
Surjections, Injections, Bijections

$$f \text{ is a surjection from } S \text{ to } T \iff \operatorname{ran} f = T$$

$$f \text{ is a injection from } S \text{ to } T \iff f^{\dagger} \in T \underset{\text{PFUN}}{\longrightarrow} S$$

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

$$\iff f \text{ is an isomorphism from } S \text{ to } T$$



Back to Predicate Logic

$$
\begin{aligned}
\mathit{intexp} ::= {} & \mathsf{0} \mid \mathsf{1} \mid \ldots \\
& \mid \mathit{var} \\
& \mid -\mathit{intexp} \mid \mathit{intexp} + \mathit{intexp} \mid \mathit{intexp} - \mathit{intexp} \mid \ldots \\
\mathit{assert} ::= {} & \mathbf{true} \mid \mathbf{false} \\
& \mid \mathit{intexp} = \mathit{intexp} \mid \mathit{intexp} < \mathit{intexp} \mid \mathit{intexp} \leq \mathit{intexp} \mid \ldots \\
& \mid \neg\mathit{assert} \mid \mathit{assert} \wedge \mathit{assert} \mid \mathit{assert} \vee \mathit{assert} \\
& \mid \mathit{assert} \Rightarrow \mathit{assert} \mid \mathit{assert} \Leftrightarrow \mathit{assert} \\
& \mid \forall \mathit{var}.\, \mathit{assert} \mid \exists \mathit{var}.\, \mathit{assert}
\end{aligned}
$$

Denotational Semantics of Predicate Logic

The meaning of term $e \in \mathit{intexp}$ is $[\![e]\!]_{\mathit{intexp}}$

i.e. the function $[\![-]\!]_{\mathit{intexp}}$ maps objects from intexp to their meanings.

What is the set of meanings?

The meaning $[\![\mathsf{5} + \mathsf{37}]\!]_{\mathit{intexp}}$ of the term $\mathsf{5} + \mathsf{37}$ could be the integer 42.

But the term $\mathsf{x} + \mathsf{5}$ contains the free variable x...

Environments

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

$$\sigma \in \Sigma \stackrel{\text{def}}{=} var \to \mathbf{Z}$$

to give meaning to free variables.

The meaning of a term is a function from the states to $\mathbf{Z}$ or $\mathbf{B}$.

$$\begin{array}{lcl} [\![-]\!]_{intexp} & \in & intexp \to \Sigma \to \mathbf{Z} \\ [\![-]\!]_{assert} & \in & assert \to \Sigma \to \mathbf{B} \end{array}$$

$$\text{if } \sigma = [\mathsf{x} : 3, \mathsf{y} : 4], \quad \text{then} \quad \begin{array}{l} [\![\mathsf{x+5}]\!]_{intexp}\, \sigma = 8 \\ [\![\exists \mathsf{z}.\, \mathsf{x} < \mathsf{z} \wedge \mathsf{z} < \mathsf{y}]\!]\, \sigma = \mathbf{false} \end{array}$$

Direct Semantics Equations for Predicate Logic

$v \in var \qquad e \in intexp \qquad p \in assert$

$$
\begin{aligned}
[\![0]\!]_{intexp}\sigma &= 0 \\
(\text{ really } [\![c_0\langle\rangle]\!]_{intexp}\sigma &= 0) \\
[\![v]\!]_{intexp}\sigma &= \sigma v \\
[\![e_0 + e_1]\!]_{intexp}\sigma &= [\![e_0]\!]_{intexp}\sigma + [\![e_1]\!]_{intexp}\sigma \\
[\![\mathbf{true}]\!]_{assert}\sigma &= \mathbf{true} \\
[\![e_0 = e_1]\!]_{assert}\sigma &= [\![e_0]\!]_{intexp}\sigma = [\![e_1]\!]_{intexp}\sigma \\
[\![\neg p]\!]_{assert}\sigma &= \neg([\![p]\!]_{assert}\sigma) \\
[\![p_0 \wedge p_1]\!]_{assert}\sigma &= [\![p_0]\!]_{assert}\sigma \wedge [\![p_1]\!]_{assert}\sigma \\
[\![\forall v.\, p]\!]_{assert}\sigma &= \forall n \in \mathbf{Z}.\, [\![p]\!]_{assert}[\sigma|v:n]
\end{aligned}
$$

Example: The Meaning of a Term

$$\begin{aligned}
&[\![\forall \mathsf{x}.\ \mathsf{x{+}0{=}x}]\!]_{assert}\sigma \\
&\quad = \forall n \in \mathbf{Z}.\ [\![\mathsf{x{+}0{=}x}]\!]_{assert}[\sigma|\mathsf{x} : n] \\
&\quad = \forall n \in \mathbf{Z}.\ [\![\mathsf{x{+}0}]\!]_{intexp}[\sigma|\mathsf{x} : n] = [\![\mathsf{x}]\!]_{intexp}[\sigma|\mathsf{x} : n] \\
&\quad = \forall n \in \mathbf{Z}.\ [\![\mathsf{x}]\!]_{intexp}[\sigma|\mathsf{x} : n] + [\![\mathsf{0}]\!]_{intexp}[\sigma|\mathsf{x} : n] = [\![\mathsf{x}]\!]_{intexp}[\sigma|\mathsf{x} : n] \\
&\quad = \forall n \in \mathbf{Z}.\ [\sigma|\mathsf{x} : n](\mathsf{x}) + 0 = [\sigma|\mathsf{x} : n](\mathsf{x}) \\
&\quad = \forall n \in \mathbf{Z}.\ n + 0 = n \\
&\quad = \mathbf{true}
\end{aligned}$$

Properties of the Semantic Equations

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

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

- They define compositional semantic functions

  (depending only on the meaning of the subterms)

  $\Rightarrow$ "equivalent" subterms can be substituted

Validity of Assertions

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

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

p is unsatisfiable $\iff \forall\sigma \in \Sigma.\ [\![p]\!]_{assert}\sigma = \mathbf{false}$

$\iff \neg p$ 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'

and p' is stronger than p

Inference Rules

Class	*Examples*
$\vdash p$ (Axiom)	$\vdash \mathsf{x} + \mathsf{0} = \mathsf{x}$ (xPlusZero)
$\dfrac{}{\vdash p}$ (Axiom Schema)	$\dfrac{}{\vdash e_1 = e_0 \Rightarrow e_0 = e_1}$ (SymmObjEq)
$\dfrac{\vdash p_0 \quad \ldots \quad \vdash p_{n-1}}{\vdash p}$ (Rule)	$\dfrac{\vdash p \quad \vdash p \Rightarrow p'}{\vdash p'}$ (ModusPonens)
	$\dfrac{\vdash p}{\vdash \forall v.\, p}$ (Generalization)

Formal Proofs

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.	$\vdash \mathsf{x} + 0 = \mathsf{x}$	(xPlusZero)
2.	$\vdash \mathsf{x} + 0 = \mathsf{x} \Rightarrow \mathsf{x} = \mathsf{x} + 0$	(SymmObjEq) $[e_0 : \mathsf{x} \mid e_1 : \mathsf{x} + 0]$
3.	$\vdash \mathsf{x} = \mathsf{x} + 0$	(ModusPonens, 1, 2) $[p : \mathsf{x} + 0 = \mathsf{x} \mid p' : \mathsf{x} = \mathsf{x} + 0]$
4.	$\vdash \forall \mathsf{x}.\, \mathsf{x} = \mathsf{x} + 0$	(Generalization, 3) $[v : \mathsf{x} \mid p : \mathsf{x} = \mathsf{x} + 0]$

Tree Representation of Formal Proofs

$$\dfrac{\vdash x + 0 = x \qquad \dfrac{}{\vdash x + 0 = x \Rightarrow x = x + 0}\,(\text{SymmObjEq})}{\dfrac{\vdash x = x + 0}{\vdash \forall x.\, x = x + 0}\,(\text{Gen})}\,(\text{MP})$$

Soundness of a Logical Theory

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:

$\vdash p \Rightarrow \forall v.\, p$ is not a sound rule, although $\dfrac{\vdash p}{\vdash \forall v.\, p}$ is.

Completeness of a Logical Theory

A logical theory $\vdash$ is complete if

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

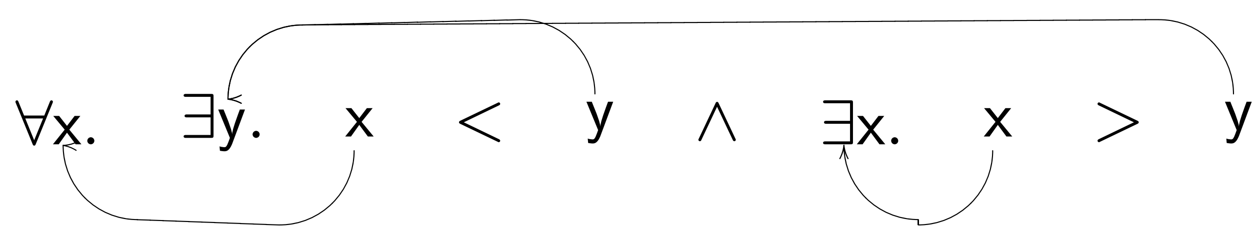
A logical theory $\vdash$ is axiomatizable if

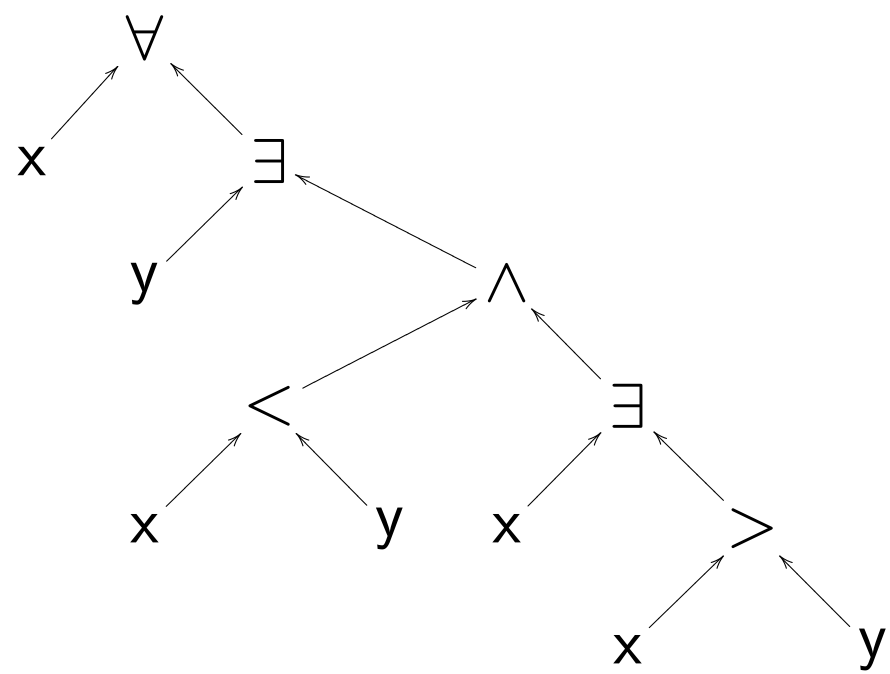
there exists a finite set of inference rules

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

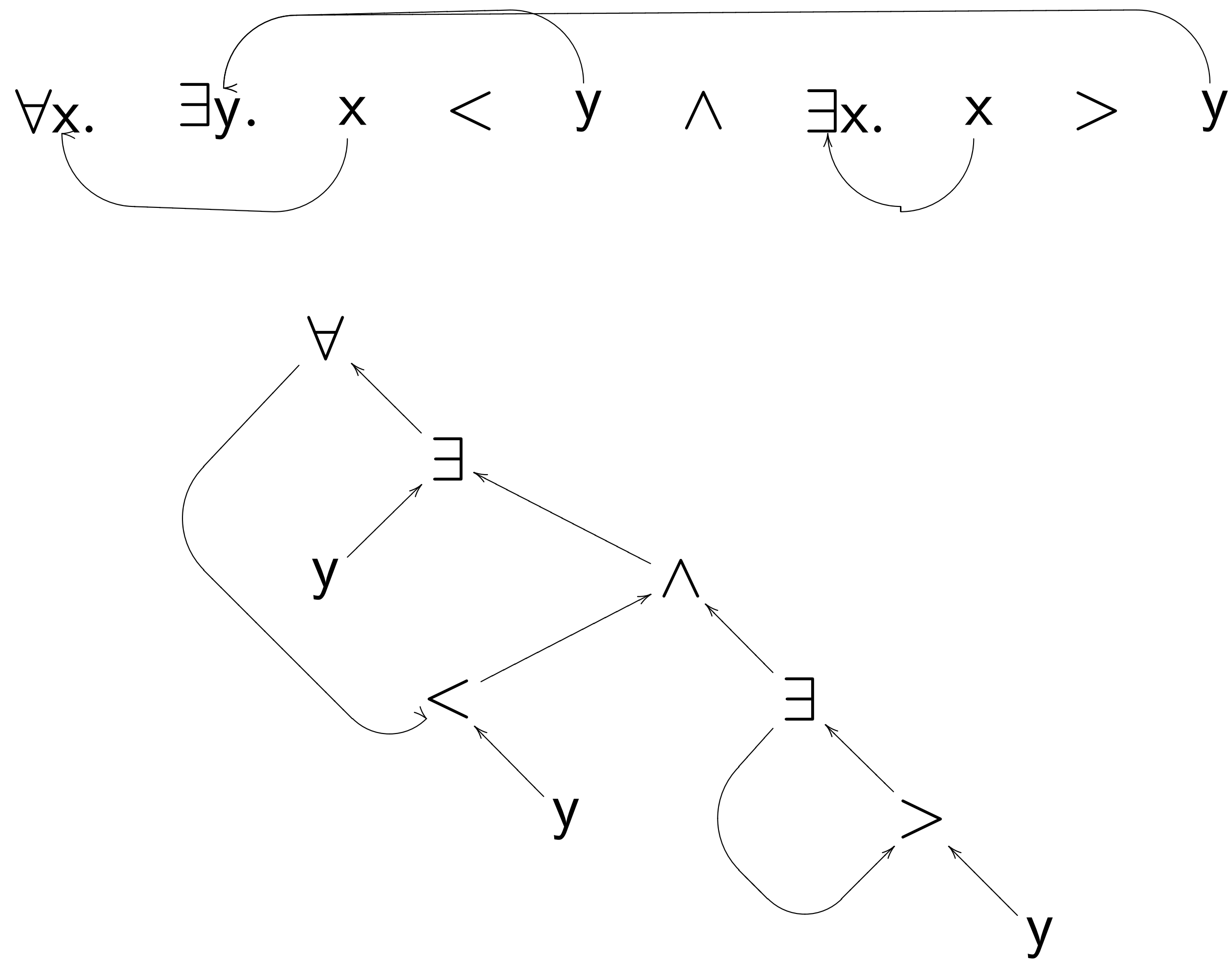
No first-order theory of arithmetic is complete and axiomatizable.

Variable Binding





Variable Binding



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.

$$
\begin{aligned}
FV_{intexp}(\mathsf{0}) &= \{\} \\
FV(v) &= \{v\} \\
FV(-e) &= FV(e) \\
FV(e_0 + e_1) &= FV(e_0) \cup FV(e_1)
\end{aligned}
\qquad
\begin{aligned}
FV_{assert}(\mathbf{true}) &= \{\} \\
FV(e_0 \mathsf{=} e_1) &= FV(e_0) \cup FV(e_1) \\
FV(\neg p) &= FV(p) \\
FV(p_0 \wedge p_1) &= FV(p_0) \cup FV(p_1) \\
FV(\forall v.\, p) &= FV(p) - \{v\}
\end{aligned}
$$

Example:

$$FV(\exists \mathsf{y}.\, \mathsf{x} < \mathsf{y} \wedge \exists \mathsf{x}.\, \mathsf{x} > \mathsf{y}) = \{\mathsf{x}\}$$

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 θ).

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:

$p' = \mathsf{0} \;\Rightarrow\; [\![\mathsf{0}]\!]_{intexp}\sigma = 0 = [\![\mathsf{0}]\!]_{intexp}\sigma'$

$p' = v \;\Rightarrow\; [\![v]\!]_{intexp}\sigma = \sigma\, v = \sigma' v = [\![v]\!]_{intexp}\sigma'$, since $FV(v) = \{v\}$.

Proof of Concidence 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 $[\![e_i]\!]_{intexp}\sigma = [\![e_i]\!]_{intexp}\sigma', i \in \{1, 2\}$.

$$\begin{aligned} [\![p']\!]_{intexp}\sigma &= [\![e_0]\!]_{intexp}\sigma + [\![e_1]\!]_{intexp}\sigma \\ &= [\![e_0]\!]_{intexp}\sigma' + [\![e_1]\!]_{intexp}\sigma' = [\![p']\!]_{intexp}\sigma' \end{aligned}$$

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

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

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

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

$$[\![\forall u.\, q]\!]_{assert}\sigma = [\![\forall u.\, q]\!]_{assert}\sigma'.$$

Substitution

$$\left.\begin{array}{l} -/\delta \in \mathit{intexp} \to \mathit{intexp} \\ -/\delta \in \mathit{assert} \to \mathit{assert} \end{array}\right\} \text{when } \delta \in \mathit{var} \to \mathit{intexp}$$

$$\begin{aligned} \mathsf{0}/\delta &= \mathsf{0} \\ (\mathsf{-}e)/\delta &= \mathsf{-}(e/\delta) \\ (e_0\mathsf{+}e_1)/\delta &= (e_0/\delta)\mathsf{+}(e_1/\delta) \\ &\dots \end{aligned}$$

$$\begin{aligned} v/\delta &= \delta v \\ (p_0 \land p_1)/\delta &= (p_0/\delta) \land (p_1/\delta) \\ (\forall v.\, p)/\delta &= \forall v'.\, (p/[\delta|v : v']), \\ \text{where } v' &\notin \bigcup_{u \in FV(p)-\{v\}} FV(\delta u) \end{aligned}$$

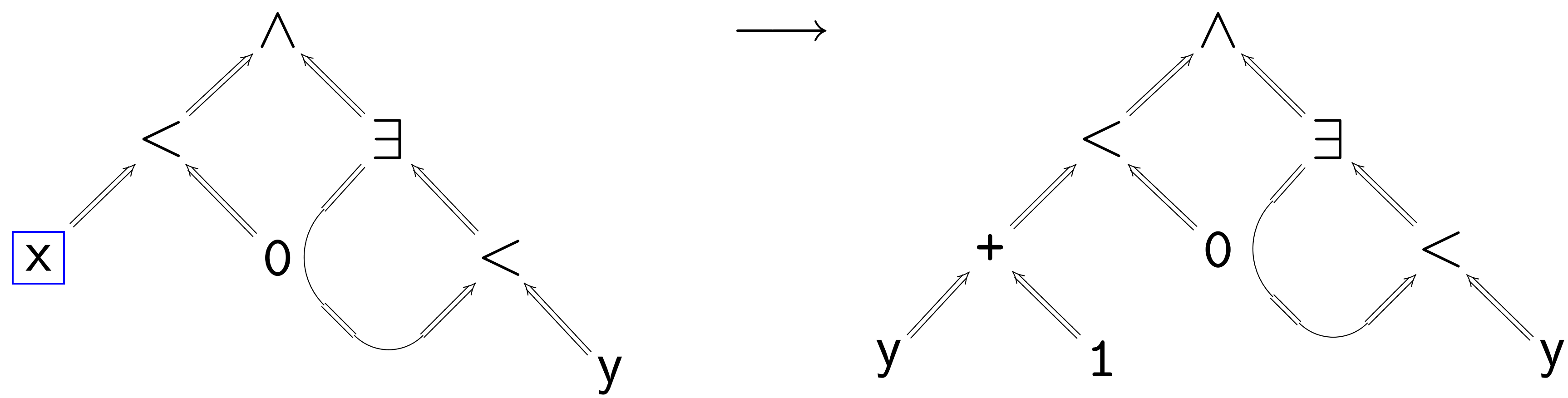
Examples:

$$(\mathsf{x} < \mathsf{0} \land \exists\mathsf{x}.\, \mathsf{x} \leq \mathsf{y})/[\mathsf{x} : \mathsf{y+1}] = \mathsf{y+1} < \mathsf{0} \land \exists\mathsf{x}.\, \mathsf{x} \leq \mathsf{y}$$

$$(\mathsf{x} < \mathsf{0} \land \exists\mathsf{x}.\, \mathsf{x} \leq \mathsf{y})/[\mathsf{y} : \mathsf{x+1}] = \mathsf{x} < \mathsf{0} \land \exists\mathsf{z}.\, \mathsf{z} \leq \mathsf{x+1}$$

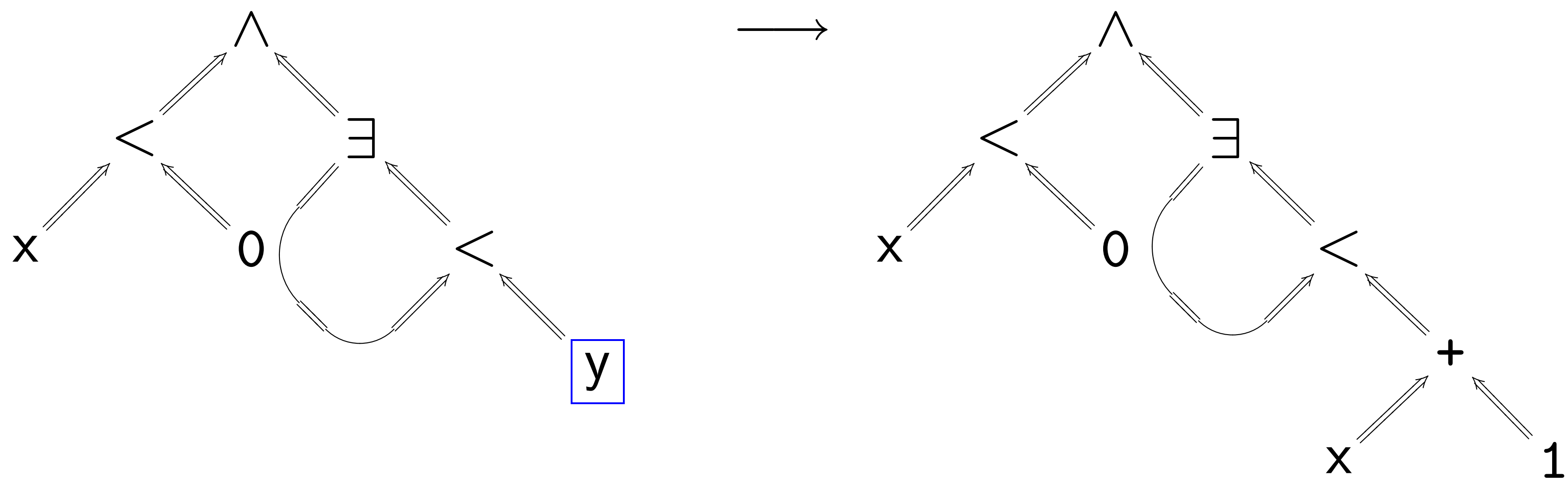
Preserving Binding Structure

$$(\mathsf{x} < 0 \land \exists \mathsf{x}.\, \mathsf{x} \le \mathsf{y})/[\boxed{\mathsf{x}} : \mathsf{y{+}1}] \quad = \quad \mathsf{y{+}1} < 0 \land \exists \mathsf{x}.\, \mathsf{x} \le \mathsf{y}$$



Avoiding Variable Capture

$$(x < 0 \land \exists x.\, x \leq y)/[y : x{+}1] \quad = \quad x < 0 \land \exists z.\, z \leq x{+}1$$



Substitution Theorems

Substitution Theorem:

If $\sigma = [\![-]\!]_{intexp}\sigma' \cdot \delta$ on $FV(p)$, then $([\![-]\!]\sigma)p = ([\![-]\!]\sigma' \cdot (-/\delta))p$.

Finite Substitution Theorem:

$$[\![p/v_0 \to e_0, \ldots v_{n-1} \to e_{n-1}]\!]\sigma = [\![p]\!][\sigma|v_0 : [\![e_0]\!]\sigma, \ldots].$$

where

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

Renaming:

If $u \notin FV(q) - \{v\}$, then $[\![\forall u.\, (q/v \to u)]\!]_{boolexp} = [\![\forall v.\, q]\!]_{boolexp}$.