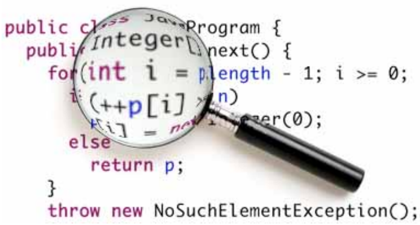
15-819M: Data, Code, Decisions

08: First-Order Logic

André Platzer

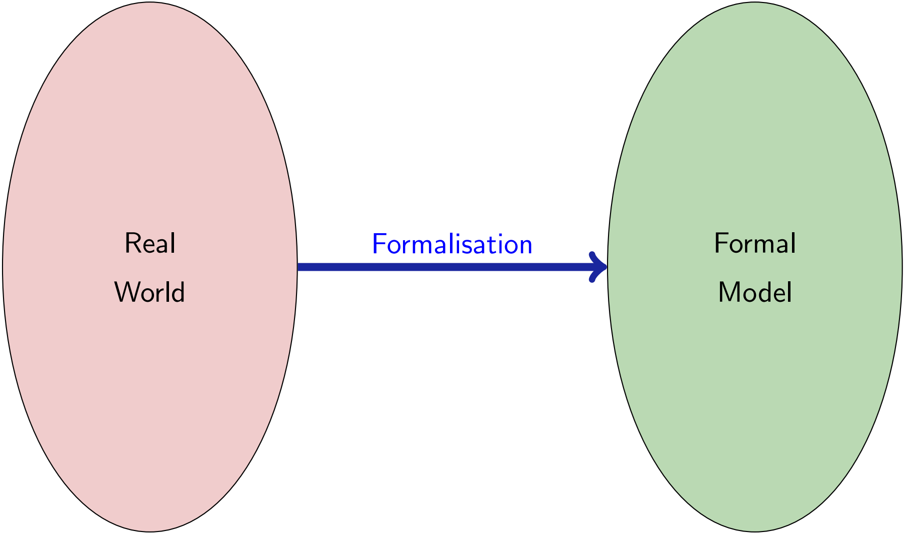
aplatzer@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA



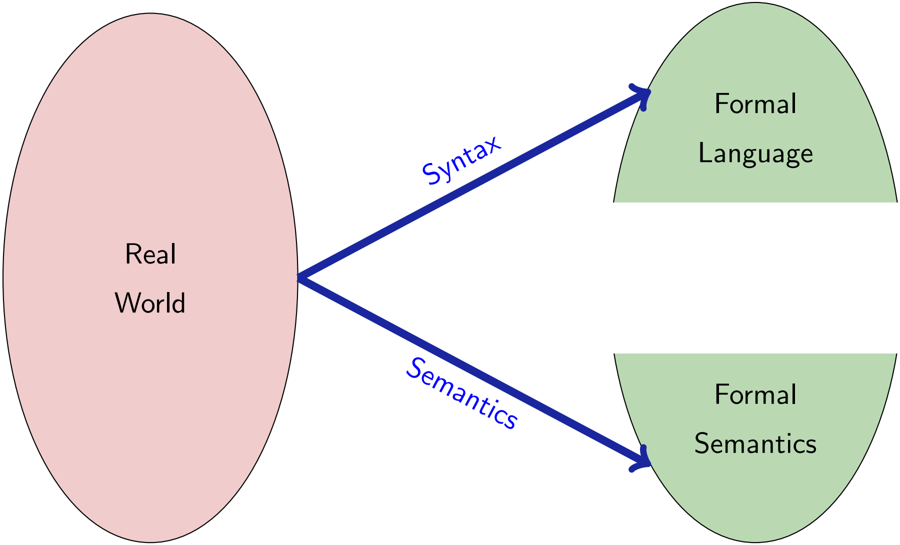
Outline

Outline

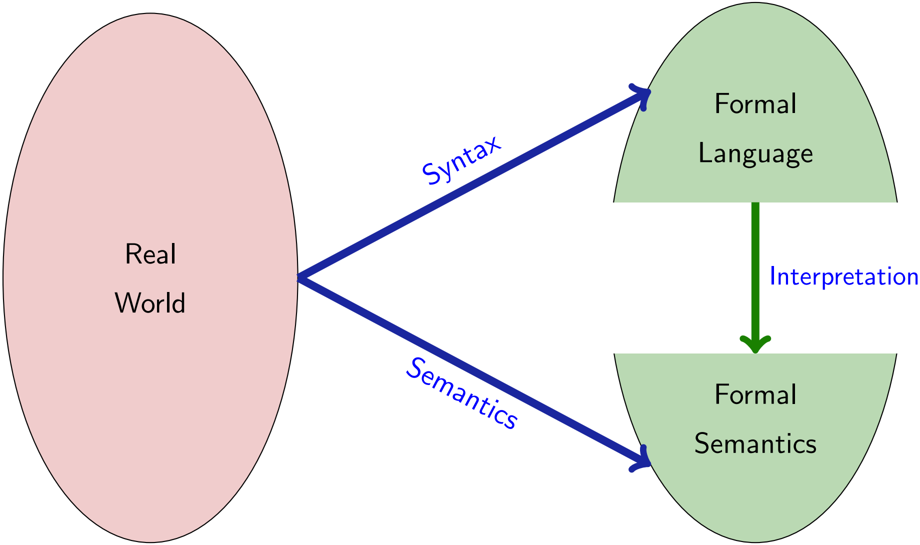
Formalisation



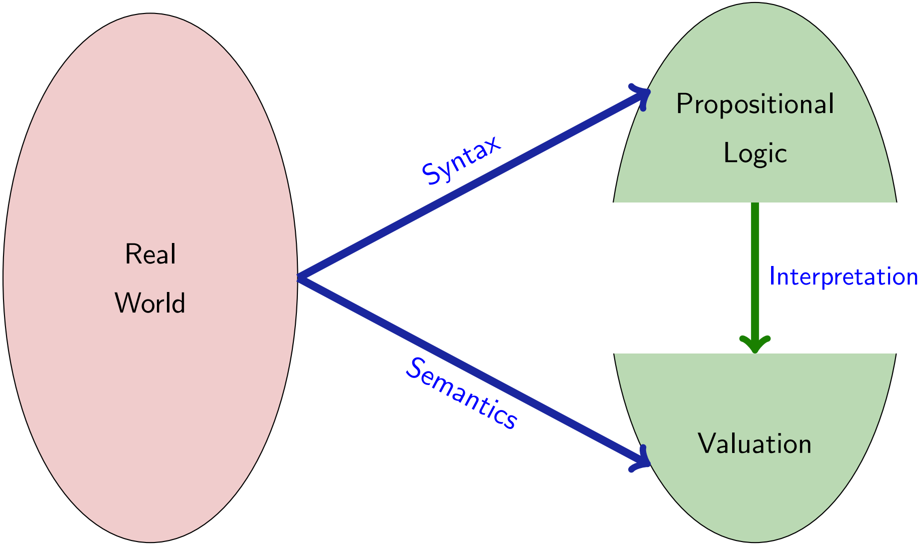
Formalisation: Syntax, Semantics



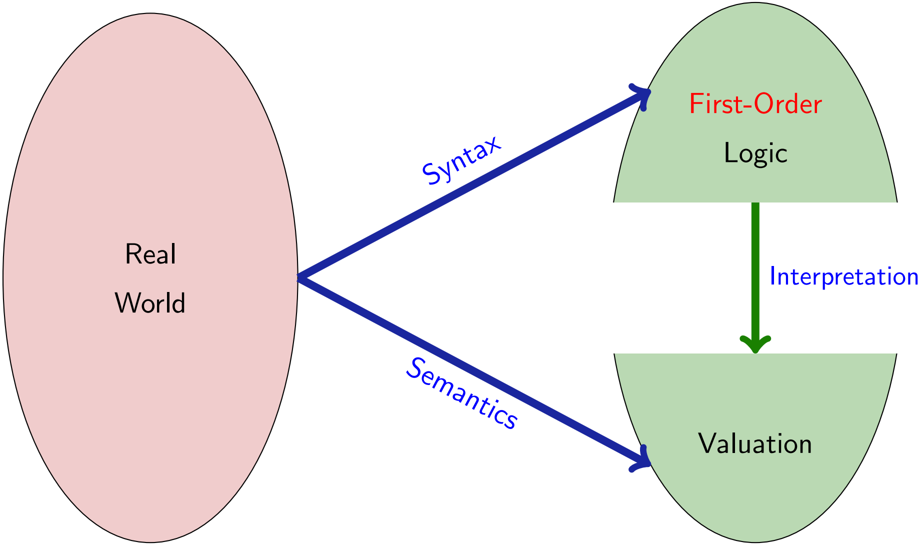
Formalisation: Syntax, Semantics



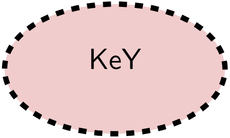
Formalisation: Syntax, Semantics



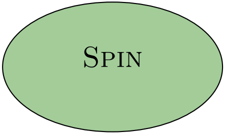
Formalisation: Syntax, Semantics



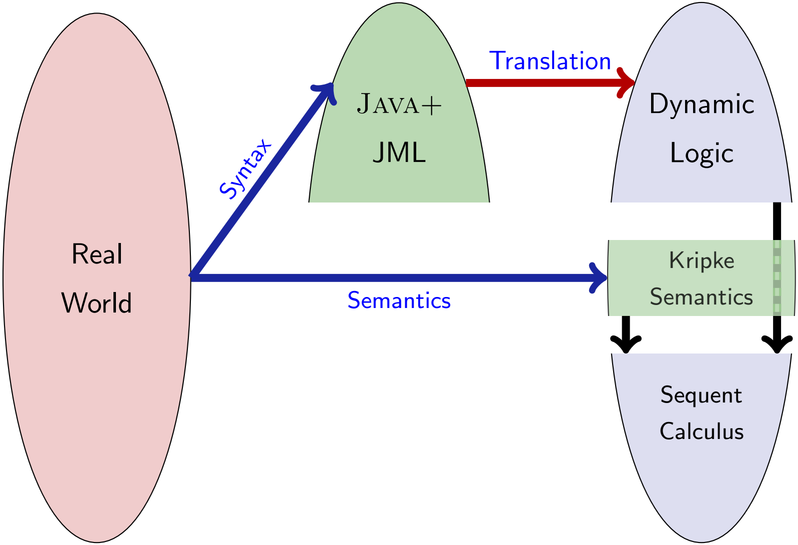
Approaches to Formal Software Verification



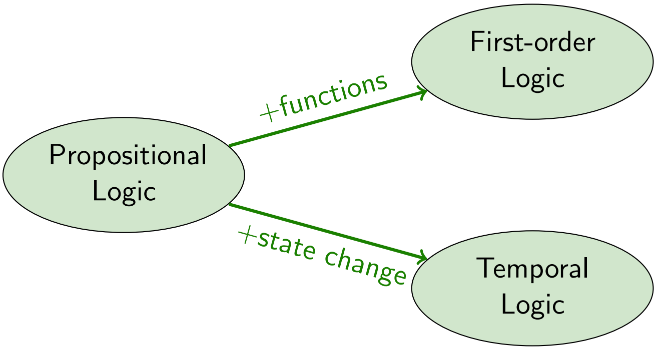
Concrete programs, Complex properties	Concrete programs, Simple properties
Abstract programs, Complex properties	Abstract programs, Simple properties



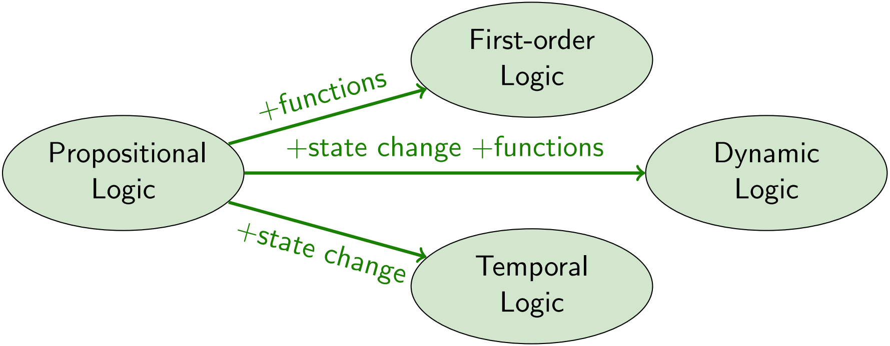
Formal Verification: Deduction



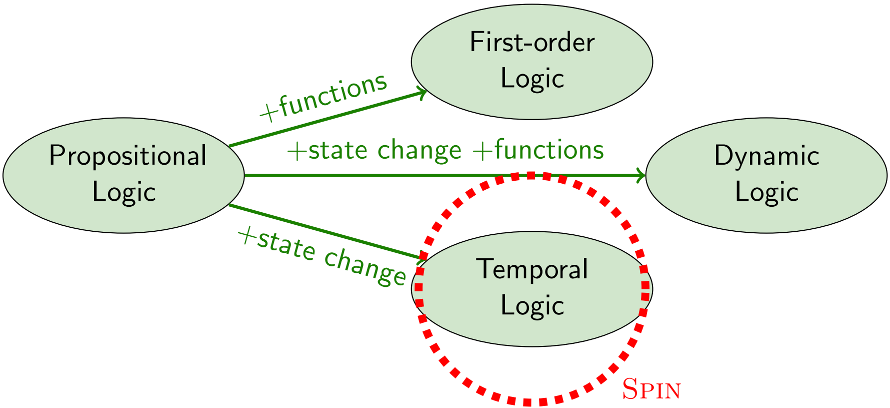
Beyond Propositional Logic



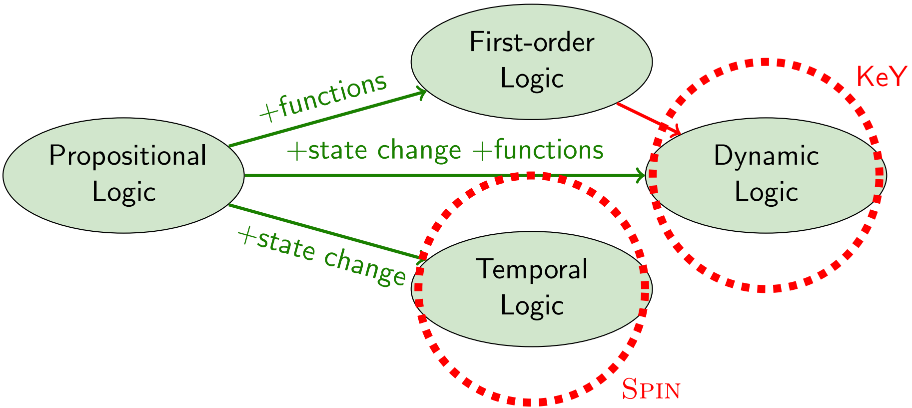
Beyond Propositional Logic



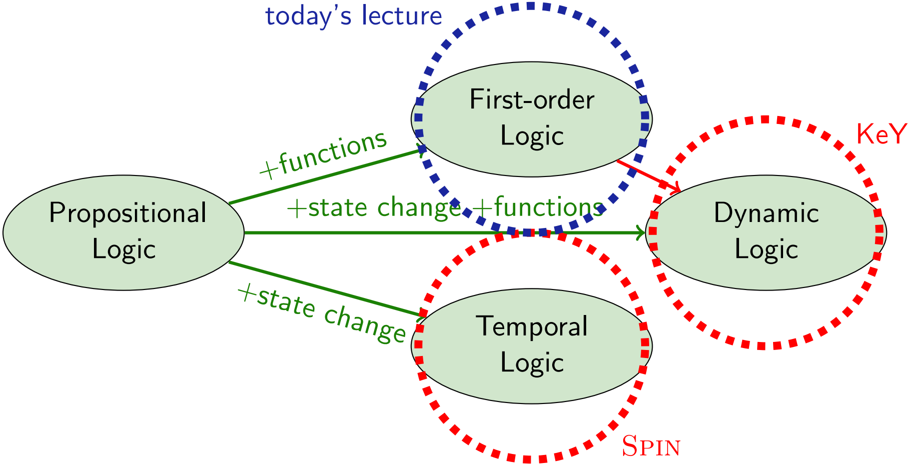
Beyond Propositional Logic



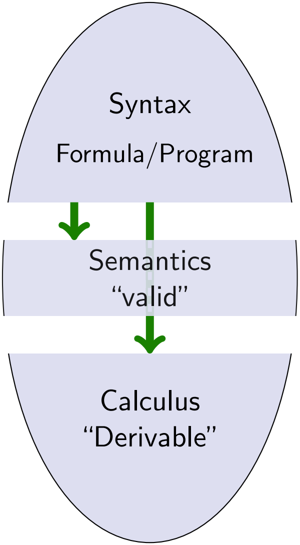
Beyond Propositional Logic



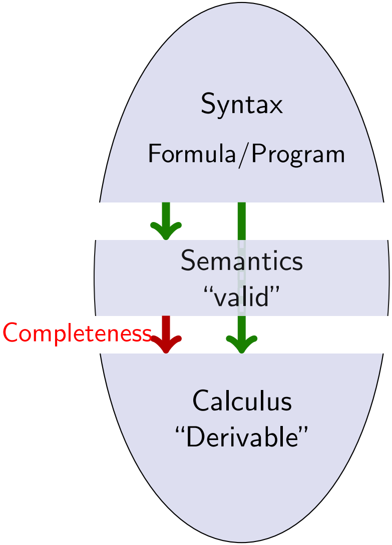
Beyond Propositional Logic



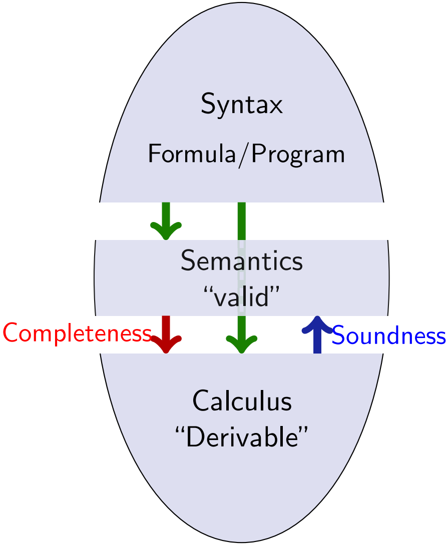
Syntax, Semantics, Calculus



Syntax, Semantics, Calculus



Syntax, Semantics, Calculus



Limitations of Propositional Logic

Fixed, finite number of objects

Cannot express: let g be group with arbitrary number of elements

No functions or relations with arguments

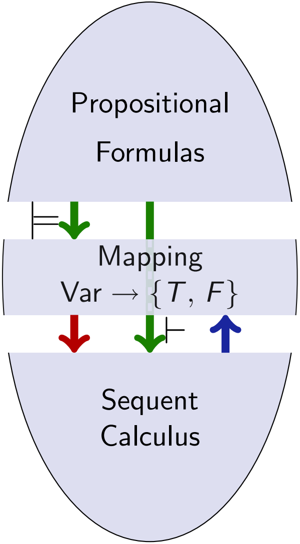
- ✔ Can express: finite function/relation table with indexed variables p_{ij}
- ✘ Cannot express:
  properties of function/relation on all arguments: "+" associative

Static interpretation

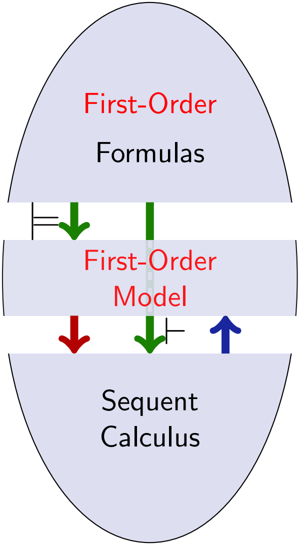
Programs change value of their variables, e.g., via assignment, call, etc.
Propositional formulas look at one single interpretation at a time

Outline

Propositional Logic



First-Order Logic



Syntax of First-Order Logic: Signature

Definition (First-Order Signature)

First-order signature $\Sigma = (\text{PSym}, \text{FSym}, \alpha)$

Predicate or Relation Symbols $\text{PSym} = \{p_i \mid i \in \mathbb{N}\}$
Function Symbols $\text{FSym} = \{f_i \mid i \in \mathbb{N}\}$

Typing function α, set of types $\mathcal{T}$

- $\alpha(p) \in \mathcal{T}^*$ for all $p \in \text{PSym}$
- $\alpha(f) \in \mathcal{T}^* \times \mathcal{T}$ for all $f \in \text{FSym}$

Definition (Variables)

$\text{VSym} = \{x_i \mid i \in \mathbb{N}\}$ set of typed variables

- In contrast to "standard" FOL, our symbols are typed
  Necessary to model a typed programming language such as JAVA!
- Allow any non-reserved name for symbols, not merely p_3, $f_{17}, \ldots$

Syntax of First-Order Logic: Signature

Declaration of signature symbols

- Write $T\ x$; to declare variable x of type T
- Write $p(T_1, \ldots, T_r)$; for $\alpha(p) = (T_1, \ldots, T_r)$
- Write $T\ f(T_1, \ldots, T_r)$; for $\alpha(f) = ((T_1, \ldots, T_r), T)$

Similar convention as in JAVA, no overloading of symbols
Case $r = 0$ is allowed, then write p instead of $p()$, etc.

Syntax of First-Order Logic: Signature

Declaration of signature symbols

- Write $T\ x$; to declare variable x of type T
- Write $p(T_1, \ldots, T_r)$; for $\alpha(p) = (T_1, \ldots, T_r)$
- Write $T\ f(T_1, \ldots, T_r)$; for $\alpha(f) = ((T_1, \ldots, T_r), T)$

Similar convention as in JAVA, no overloading of symbols
Case $r = 0$ is allowed, then write p instead of $p()$, etc.

Example

Variables **boolean** `b;` **int** `i;`

Predicates `isEmpty(List); alertOn;`

Functions **int** `arrayLookup(`**int**`); java.lang.Object o;`

OO Type Hierarchy

We want to model the behavior of JAVA programs
Admissible types $\mathcal{T}$ form object-oriented type hierarchy

OO Type Hierarchy

We want to model the behavior of JAVA programs
Admissible types $\mathcal{T}$ form object-oriented type hierarchy

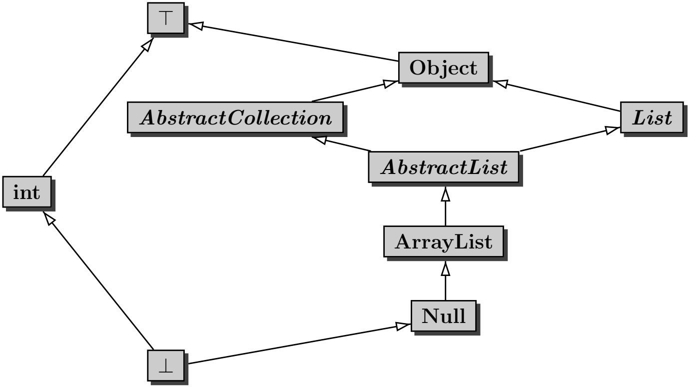
Definition (OO Type Hierarchy)

- $\mathcal{T}$ is finite set of types (not parameterized)
- Given subtype relation $\sqsubseteq$, assume $\mathcal{T}$ has all supertypes, i.e., $\mathcal{T}$ is $\sqcap$-closed
- Dynamic types $\mathcal{T}_d \subseteq \mathcal{T}$, where $\top \in \mathcal{T}_d$
- Abstract types $\mathcal{T}_a \subseteq \mathcal{T}$, where $\bot \in \mathcal{T}_a$
- $\mathcal{T}_d \cap \mathcal{T}_a = \emptyset$
- $\mathcal{T}_d \cup \mathcal{T}_a = \mathcal{T}$
- $\bot \sqsubseteq T \sqsubseteq \top$ for all $T \in \mathcal{T}$

OO Type Hierarchy

Example

Using UML notation



OO Type Hierarchy

- Dynamic types are those with direct elements
- Abstract types for abstract classes and interfaces
- JAVA 1.5+ is $\sqcap$-closed
- In JAVA primitive (value) and object types incomparable
- $\bot$ is abstract and hence no object ever can have this type
  $\bot$ cannot occur in declaration of signature symbols
- Each abstract type except $\bot$ has a non-empty dynamic subtype
- In JAVA $\top$ is chosen to have no direct elements
- JAVA has infinitely many types: **int** [], **int** [] [],...
  Restrict $\mathcal{T}$ to the finitely many types that occur in a given program

OO Type Hierarchy

- Dynamic types are those with direct elements
- Abstract types for abstract classes and interfaces
- JAVA 1.5+ is $\sqcap$-closed
- In JAVA primitive (value) and object types incomparable
- $\bot$ is abstract and hence no object ever can have this type
  $\bot$ cannot occur in declaration of signature symbols
- Each abstract type except $\bot$ has a non-empty dynamic subtype
- In JAVA $\top$ is chosen to have no direct elements
- JAVA has infinitely many types: **int** [], **int** [] [],. . .
  Restrict $\mathcal{T}$ to the finitely many types that occur in a given program

Example (The Minimal Type Hierarchy)

$\mathcal{T} = \{\bot, \top\}$

All signature symbols have same type $\top$: drop type, untyped logic

Reserved Signature Symbols

Reserved signature symbols

- Equality symbol $\doteq \in$ PSym declared as $\doteq(\top, \top)$

  Written infix: $x \doteq 0$

- Type predicate symbol $\sqsubseteq T \in$ PSym for each $T \in \mathcal{T}$

  Declared as $\sqsubseteq T(\top)$

  Written prefix: `i`$\sqsubseteq$**int** — read "instance of"

- Type cast symbol $(T) \in$ FSym for each $T \in \mathcal{T}$

  Declared as $T\ (T)(\top)$
  Written prefix: `(String)o` — read "cast o to String"

Reserved Signature Symbols

Reserved signature symbols

- Equality symbol $\doteq \in$ PSym declared as $\doteq(\top, \top)$

  Written infix: $x \doteq 0$

- Type predicate symbol $\sqsubseteq T \in$ PSym for each $T \in \mathcal{T}$

  Declared as $\sqsubseteq T(\top)$

  Written prefix: `i`$\sqsubseteq$**int** — read "instance of"

- Type cast symbol $(T) \in$ FSym for each $T \in \mathcal{T}$

  Declared as $T\ (T)(\top)$
  Written prefix: `(String)o` — read "cast o to String"

So far, we have a type system and a signature — where is the logic?

Terms

First-order terms, informally

- Think of first-order terms as expressions in a programming language
  Built up from variables, constants, function symbols
- First-order terms have no side effects (like PROMELA, unlike JAVA)
- First-order terms have a type and must respect type hierarchy
  - type of $f(g(x))$ is result type in declaration of function f
  - in $f(g(x))$ the result type of g is subtype of argument type of f, etc.

Terms

First-order terms, informally

- Think of first-order terms as expressions in a programming language
  Built up from variables, constants, function symbols
- First-order terms have no side effects (like PROMELA, unlike JAVA)
- First-order terms have a type and must respect type hierarchy
  - type of $f(g(x))$ is result type in declaration of function f
  - in $f(g(x))$ the result type of g is subtype of argument type of f, etc.

Definition (First-Order Terms $\{\mathrm{Term}_T\}_{T \in \mathcal{T}}$ with type $T \in \mathcal{T}$)

- x is term of type T for variable declared as $T\ x$;
- $f(t_1, \ldots, t_r)$ is term of type T for
  - function symbol declared as $T\ f(T_1, \ldots, T_r)$; and
  - terms t_i of type $T_i' \sqsubseteq T_i$ for $1 \leq i \leq r$
- There are no other terms (inductive definition)

Terms

Example

Signature: **int** `i`; **short** `j`; `List l`; **int** `f`(**int**);

- `f(i)` has result type **int** and is contained in $\mathrm{Term}_{\mathbf{int}}$
- `f(j)` has result type **int** (when $\mathbf{short} \sqsubseteq \mathbf{int}$)
- `f(l)` is ill-typed (when **int**, `List` incomparable)
- `f(i,i)` is not a term (doesn't match declaration)
- (**int**)`j` is term of type **int**
- even (**int**)`l` is term of type **int** (type cast always well-formed)

Terms

Example

Signature: **int** `i`; **short** `j`; `List l`; **int** `f(`**int**`)`;

- `f(i)` has result type **int** and is contained in $\text{Term}_{\textbf{int}}$
- `f(j)` has result type **int** (when **short**$\sqsubseteq$**int**)
- `f(l)` is ill-typed (when **int**, `List` incomparable)
- `f(i,i)` is not a term (doesn't match declaration)
- `(`**int**`)j` is term of type **int**
- even `(`**int**`)l` is term of type **int** (type cast always well-formed)

- If f is constant ($r = 0$) write f instead of $f()$
- Use infix notation liberally, where appropriate: declare **int** `+(`**int**`,` **int**`)`; then write `i+j`, etc.
- Use brackets to disambiguiate parsing: `(i+j)*i`

First-Order Atomic Formulas

Definition (Atomic First-Order Formulas)

$p(t_1, \ldots, t_r)$ is atomic first-order formula for

- predicate symbol declared as $p(T_1, \ldots, T_r)$; and
- terms t_i of type $T_i' \sqsubseteq T_i$ for $1 \leq i \leq r$

First-Order Atomic Formulas

Definition (Atomic First-Order Formulas)

$p(t_1, \ldots, t_r)$ is atomic first-order formula for

- predicate symbol declared as $p(T_1, \ldots, T_r)$; and
- terms t_i of type $T'_i \sqsubseteq T_i$ for $1 \leq i \leq r$

Example

Signature: **int** `i`; **short** `j`; `List l`; `<(`**int**, **int**`)`;

- `i < i` is an atomic first-order formula
- `i < j` is an atomic first-order formula (when **short**$\sqsubseteq$**int**)
- `i < l` is ill-typed (when **int**, `List` incomparable)
- `i`$\doteq$`j` and even `i`$\doteq$`l` are atomic first-order formulas
- `i`$\sqsubseteq$**short** is an atomic first-order formula

First-Order Formulas

Definition (Set of First-Order Formulas *For*)

- Truth constants true, false and all first-order atomic formulas are first-order formulas
- If ϕ and ψ are first-order formulas then

$$!\,\phi, \quad (\phi \;\&\; \psi), \quad (\phi \mid \psi), \quad (\phi \;\texttt{->}\; \psi), \quad (\phi \;\texttt{<->}\; \psi)$$

  are also first-order formulas
- If $T\ x$ is a variable declaration, ϕ a first-order formula, then $\forall\, T\ x;\ \phi$ and $\exists\, T\ x;\ \phi$ are first-order formulas
  Any occurrence of x in ϕ must be well-typed

- $\forall\, T\ x;\ \phi$ called universally quantified formula
- $\exists\, T\ x;\ \phi$ called existentially quantified formula

First-Order Formulas

- In $\forall\ T\ x;\ \phi$ and $\exists\ T\ x;\ \phi$ call ϕ the scope of x bound by $\forall/\exists$
- Variables bound in quantified formulas similar to program locations declared as local variables/formal parameters

Example

- $\forall\,\mathbf{int}\ i;\ \exists\,\mathbf{int}\ j;\ i < j$ is a first-order formula
- $\forall\,\mathbf{int}\ i;\ \exists\,\texttt{List}\ l;\ i < l$ is ill-typed
- $\forall\,\mathbf{int}\ i;\ i < j$ is a first-order formula if j is a constant compatible with $\mathbf{int}$
- $(\forall\,\mathbf{int}\ i;\ \forall\,\mathbf{int}\ j;\ i < j)\ \mid\ (\forall\,\mathbf{int}\ i;\ \forall\,\mathbf{int}\ j;\ i > j)$ is a first-order formula

Remark on Concrete Syntax

	Text book	SPIN	KeY	JAVA
Negation	$\neg$	`!`	`!`	`!`
Conjunction	$\wedge$	`&&`	`&`	`&&`
Disjunction	$\vee$	$\vert\vert$	$\vert$	$\vert\vert$
Implication	$\rightarrow$, $\supset$	`->`	`->`	n/a
Equivalence	$\leftrightarrow$	`<->`	`<->`	n/a
Universal Quantifier	$\forall x; \phi$	n/a	`\forall` $T\ x; \phi$	n/a
Existential Quantifier	$\exists x; \phi$	n/a	`\exists` $T\ x; \phi$	n/a
Value equality	$\doteq$	`==`	`=`	`==`

Remark on Concrete Syntax

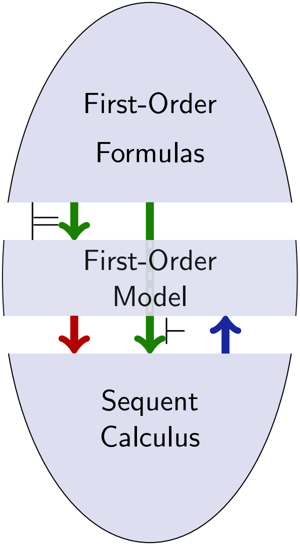
	Text book	SPIN	KeY	JAVA
Negation	$\neg$	!	!	!
Conjunction	$\wedge$	&&	&	&&
Disjunction	$\vee$	||	|	||
Implication	$\rightarrow$, $\supset$	->	->	n/a
Equivalence	$\leftrightarrow$	<->	<->	n/a
Universal Quantifier	$\forall x; \phi$	n/a	`\forall` $T\ x; \phi$	n/a
Existential Quantifier	$\exists x; \phi$	n/a	`\exists` $T\ x; \phi$	n/a
Value equality	$\doteq$	==	=	==

For quantifiers we normally use textbook syntax and suppress type information to ease readability

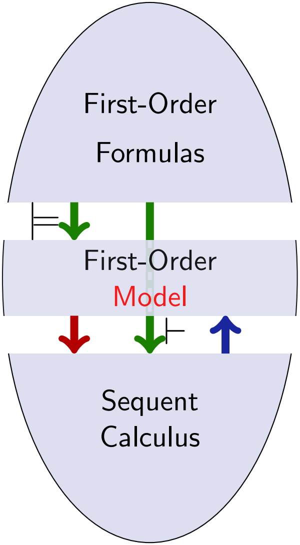
For propositional connectives we use KeY syntax

Outline

First-Order Semantics



First-Order Semantics



First-Order Semantics

From propositional to first-order semantics

- In prop. logic, interpretation of variables with $\{T, F\}$
- In first-order logic we must assign meaning to:
  - variables bound in quantifiers
  - constant and function symbols
  - predicate symbols
- Each variable or function value may denote a different object
- Respect typing: **int** `i`, `List l` must denote different objects

What we need (to interpret a first-order formula)

1. A collection of typed universes of objects (akin to heap objects)
2. A mapping from variables to objects
3. A mapping from function arguments to function values
4. The set of argument tuples where a predicate is true

First-Order Domains/Universes

1. A collection of typed universes of objects

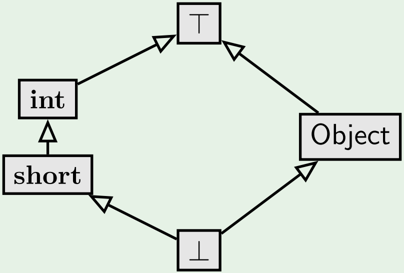
Definition (Universe/Domain)

A non-empty set $\mathcal{D}$ of objects is a universe or domain
Each element of $\mathcal{D}$ has a fixed type given by $\delta : \mathcal{D} \to \mathcal{T}_d$

- Like heap objects and values in JAVA
- Notation for the domain elements type-compatible with $T \in \mathcal{T}$:
  $\mathcal{D}^T = \{d \in \mathcal{D} \mid \delta(d) \sqsubseteq T\}$
- For each dynamic type $T \in \mathcal{T}_d$ there must be at least one domain element type-compatible with it: $\mathcal{D}^T \neq \emptyset$

First-Order Universes

Example



- $\mathcal{D} = \{17, o\}$
- $\delta(17) = \textbf{short}$, $\delta(o) = \text{Object}$
- Then $\mathcal{D}^{\textbf{short}} = \mathcal{D}^{\textbf{int}} = \{17\}$, $\mathcal{D}^{\text{Object}} = \{o\}$, $\mathcal{D}^{\top} = \mathcal{D} = \{17, o\}$, and $\mathcal{D}^{\bot} = \{\}$

First-Order Models

3. A mapping from function arguments to function values
4. The set of argument tuples where a predicate is true

Definition (First-Order Model)

Let $\mathcal{D}$ be a domain with typing function δ

Let f be declared as $T\ f(T_1, \ldots, T_r)$;

Let p be declared as $p(T_1, \ldots, T_r)$;

Let $\mathcal{I}(f) : \mathcal{D}^{T_1} \times \cdots \times \mathcal{D}^{T_r} \rightarrow \mathcal{D}^{T}$

Let $\mathcal{I}(p) \subseteq \mathcal{D}^{T_1} \times \cdots \times \mathcal{D}^{T_r}$

Then $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ is a first-order model

First-Order Models

Example

Signature: `int i; short j; int f(int); Object obj; <(int,int);`

$\mathcal{D} = \{17, 2, o\}$ where all numbers are short

$$\mathcal{I}(i) = 17$$
$$\mathcal{I}(j) = 17$$
$$\mathcal{I}(\texttt{obj}) = o$$

$\mathcal{D}^{\textbf{int}}$	$\mathcal{I}(f)$
2	2
17	2

$\mathcal{D}^{\textbf{int}} \times \mathcal{D}^{\textbf{int}}$	in $\mathcal{I}(<)$?
$(2,2)$	F
$(2,17)$	T
$(17,2)$	F
$(17,17)$	F

One of uncountably many possible first-order models!

Semantics of Reserved Signature Symbols

Definition

- Equality symbol $\doteq$ declared as $\doteq(\top, \top)$

  Model is fixed as $\mathcal{I}(\doteq) = \{(d, d) \mid d \in \mathcal{D}\}$
  "Referential Equality" (holds if arguments refer to identical object)

  Exercise: write down the predicate table for example domain

- Type predicate symbol $\sqsubseteq T$ for any T, declared as $\sqsubseteq T(\top)$

  $\mathcal{I}(\sqsubseteq T) = \mathcal{D}^T$

  Exercise: what is $\mathcal{I}(\sqsubseteq\text{Object})$?

- Type cast symbol (T) for each T, declared as $T\ (T)(\top)$

$$\mathcal{I}((T))(x) = \begin{cases} x & \text{if cast succeeds } (\delta(x) \sqsubseteq T) \\ d & \text{otherwise, for an arbitrary fixed } d \in \mathcal{D}^T \end{cases}$$

  Exercise: what is $\mathcal{I}((\mathbf{int}))(17)$?

Signature Symbols vs. Domain Elements

- Domain elements are not just the terms representing them
- First-order formulas and terms have no access to domain
- As in JAVA: identity and memory layout of values/objects hidden
- Think of a first-order model as a “heap” of first-order logic

Example

Signature: `Object obj1, obj2;`
Domain: $\mathcal{D} = \{o\}$

In this model, necessarily $\mathcal{I}(\texttt{obj1}) = \mathcal{I}(\texttt{obj2}) = o$

Effect similar to aliasing in JAVA with reference types

Variable Assignments

2. A mapping from variables to objects

Think of variable assignment as environment for storage of local variables

Definition (Variable Assignment)

A variable assignment β maps variables to domain elements
It respects the variable type, i.e., if x has type T then $\beta(x) \in \mathcal{D}^T$

Definition (Modified Variable Assignment)

Let y be variable of type T, β variable assignment, $d \in \mathcal{D}^T$:

$$\beta_y^d(x) := \begin{cases} \beta(x) & \text{if } x \neq y \\ d & \text{if } x = y \end{cases}$$

Semantic Evaluation of Terms

Given a first-order model $\mathcal{M}$ and a variable assignment β
it is possible to evaluate first-order terms under $\mathcal{M}$ and β

Analogy

Evaluating an expression in a programming language
with respect to a given heap ($\mathcal{M}$) and binding of local variables (β)

Definition (Valuation of Terms)

$val_{\mathcal{M},\beta} : \text{Term} \rightarrow \mathcal{D}$ such that $val_{\mathcal{M},\beta}(t) \in \mathcal{D}^T$ for $t \in \text{Term}_T$:

- $val_{\mathcal{M},\beta}(x) = \beta(x)$ (recall that β respects typing)
- $val_{\mathcal{M},\beta}(f(t_1, \ldots, t_r)) = \mathcal{I}(f)(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_r))$

Semantic Evaluation of Terms

Example

Signature: **int** `i`; **short** `j`; **int** `f`(**int**);

$\mathcal{D} = \{17, 2, o\}$ where all numbers are short

Variables: `Object obj`; **int** `x`;

$\mathcal{I}(\texttt{i}) = 17$
$\mathcal{I}(\texttt{j}) = 17$

$\mathcal{D}^{\textbf{int}}$	$\mathcal{I}(\texttt{f})$
2	17
17	2

Var	β
obj	o
x	17

- $val_{\mathcal{M},\beta}(\texttt{f}(\texttt{f}(\texttt{i})))$?
- $val_{\mathcal{M},\beta}(x)$?
- $val_{\mathcal{M},\beta}((\textbf{int})\texttt{obj})$?

Semantic Evaluation of Terms

Example

Signature: **int** `i`; **short** `j`; **int** `f`(**int**);

$\mathcal{D} = \{17, 2, o\}$ where all numbers are short

Variables: `Object obj`; **int** `x`;

$\mathcal{I}(\texttt{i}) = 17$
$\mathcal{I}(\texttt{j}) = 17$

$\mathcal{D}^{\textbf{int}}$	$\mathcal{I}(\texttt{f})$
2	17
17	2

Var	β
`obj`	o
`x`	17

- $val_{\mathcal{M},\beta}(\texttt{f(f(i))})$? $= 17$
- $val_{\mathcal{M},\beta}(x)$?
- $val_{\mathcal{M},\beta}((\textbf{int})\texttt{obj})$?

Semantic Evaluation of Terms

Example

Signature: **int** `i`; **short** `j`; **int** `f(`**int**`)`;

$\mathcal{D} = \{17, 2, o\}$ where all numbers are short

Variables: `Object obj`; **int** `x`;

$\mathcal{I}(\mathtt{i}) = 17$
$\mathcal{I}(\mathtt{j}) = 17$

$\mathcal{D}^{\textbf{int}}$	$\mathcal{I}(\mathtt{f})$
2	17
17	2

Var	β
obj	o
x	17

- $val_{\mathcal{M},\beta}(\mathtt{f}(\mathtt{f}(\mathtt{i})))$? $= 17$
- $val_{\mathcal{M},\beta}(x)$? $= 17$
- $val_{\mathcal{M},\beta}((\textbf{int})\mathtt{obj})$?

Semantic Evaluation of Terms

Example

Signature: **int** `i`; **short** `j`; **int** `f`(**int**);

$\mathcal{D} = \{17, 2, o\}$ where all numbers are short

Variables: `Object obj`; **int** `x`;

$\mathcal{I}(\mathtt{i}) = 17$
$\mathcal{I}(\mathtt{j}) = 17$

$\mathcal{D}^{\textbf{int}}$	$\mathcal{I}(\mathtt{f})$
2	17
17	2

Var	β
obj	o
x	17

- $val_{\mathcal{M},\beta}(\mathtt{f}(\mathtt{f}(\mathtt{i})))\ ?\ = 17$
- $val_{\mathcal{M},\beta}(x)\ ?\ = 17$
- $val_{\mathcal{M},\beta}((\textbf{int})\mathtt{obj})\ ?\ = 2$, say

Semantic Evaluation of Formulas

Formulas are true or false

A validity relation is more convenient than a function

Definition (Validity Relation for Formulas)

$\mathcal{M}, \beta \models \phi$ for $\phi \in For$ "$\mathcal{M}, \beta$ models ϕ"

- $\mathcal{M}, \beta \models p(t_1, \ldots, t_r)$ iff $(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_r)) \in \mathcal{I}(p)$
- $\mathcal{M}, \beta \models \phi \ \& \ \psi$ iff $\mathcal{M}, \beta \models \phi$ and $\mathcal{M}, \beta \models \psi$
- ... as in propositional logic
- $\mathcal{M}, \beta \models \forall\, T\ x; \phi$ iff $\mathcal{M}, \beta_x^d \models \phi$ for all $d \in \mathcal{D}^T$
- $\mathcal{M}, \beta \models \exists\, T\ x; \phi$ iff $\mathcal{M}, \beta_x^d \models \phi$ for at least one $d \in \mathcal{D}^T$

Semantic Evaluation of Formulas

Example

Signature: `short j; int f(int); Object obj; <(int,int);`

$\mathcal{D} = \{17,\ 2,\ o\}$ where all numbers are short

$\mathcal{I}(j) = 17$
$\mathcal{I}(\texttt{obj}) = o$

$\mathcal{D}^{\mathbf{int}}$	$\mathcal{I}(f)$
2	2
17	2

$\mathcal{D}^{\mathbf{int}} \times \mathcal{D}^{\mathbf{int}}$	in $\mathcal{I}(<)$?
$(2, 2)$	F
$(2, 17)$	T
$(17, 2)$	F
$(17, 17)$	F

- $\mathcal{M}, \beta \models f(j) < j$?
- $\mathcal{M}, \beta \models \exists\, \mathbf{int}\ x;\ f(x) \doteq x$?
- $\mathcal{M}, \beta \models \forall\, \texttt{Object}\ o1;\ \forall\, \texttt{Object}\ o2;\ o1 \doteq o2$?

Semantic Notions

Definition (Satisfiability, Truth, Validity)

$$
\begin{array}{llll}
\mathcal{M}, \beta \models \phi & & & (\phi \text{ is } \textbf{satisfiable}) \\
\mathcal{M} \models \phi & \text{iff} & \text{for all } \beta : \quad \mathcal{M}, \beta \models \phi & (\phi \text{ is } \textbf{true} \text{ in } \mathcal{M}) \\
\models \phi & \text{iff} & \text{for all } \mathcal{M} : \quad \mathcal{M} \models \phi & (\phi \text{ is } \textbf{valid})
\end{array}
$$

Closed formulas that are satisfiable are also true: one top-level notion

Semantic Notions

Definition (Satisfiability, Truth, Validity)

$\mathcal{M}, \beta \models \phi$				(ϕ is **satisfiable**)
$\mathcal{M} \models \phi$	iff	for all β :	$\mathcal{M}, \beta \models \phi$	(ϕ is **true** in $\mathcal{M}$)
$\models \phi$	iff	for all $\mathcal{M}$:	$\mathcal{M} \models \phi$	(ϕ is **valid**)

Closed formulas that are satisfiable are also true: one top-level notion

Example

- $f(j) < j$ is true in $\mathcal{M}$
- $\exists\, \mathbf{int}\ x;\ i \doteq x$ is valid
- $\exists\, \mathbf{int}\ x;\ !(x \doteq x)$ is not satisfiable

Outline

Untyped First-Order Logic

Most logic textbooks introduce untyped logic

How to obtain untyped logic as a special case

- Minimal Type Hierarchy: $\mathcal{T} = \{\bot, \top\}$
- $\mathcal{D} = \mathcal{D}^\top \neq \emptyset$: only one populated type $\top$, drop all typing info
- Signature merely specifies arity of functions and predicates: Write $f/1$, $</2$, $i/0$, etc.
- Untyped logic is suitable whenever we model a uniform domain
- Typical applications: pure mathematics such as algebra

Untyped First-Order Logic

Example (Axiomatization of a group in first-order logic)

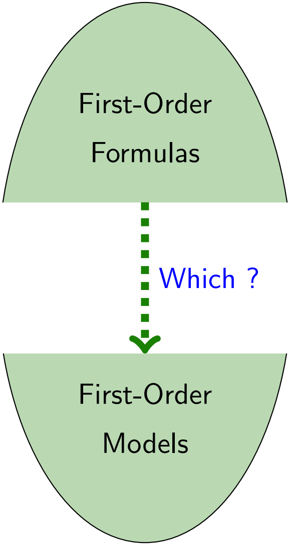
Signature Σ_G: FSym $= \{\circ/2, \mathbf{e}/0\}$, PSym $= \{\doteq/2\}$
Let G be the following formulas:

Left identity	$\forall x;\ \mathbf{e} \circ x \doteq x$
Left inverse	$\forall x;\ \exists y;\ y \circ x \doteq \mathbf{e}$
Associativity	$\forall x;\ \forall y;\ \forall z;\ (x \circ y) \circ z \doteq x \circ (y \circ z)$

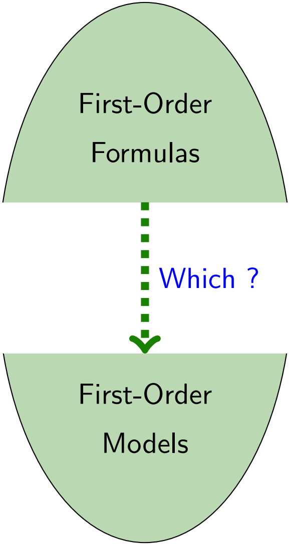
Let ϕ be Σ_G-formula.
Whenever $\models G \rightarrow \phi$, then ϕ is a theorem of group theory

Outline

Modeling with First-Order Logic

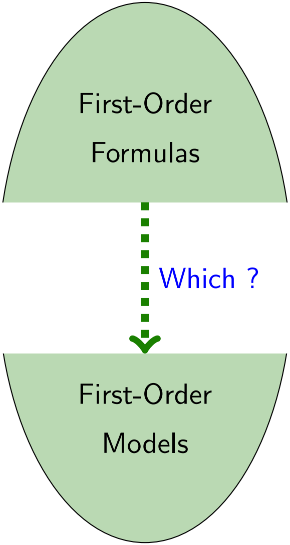


Modeling with First-Order Logic



Example (At least two elements)

Modeling with First-Order Logic

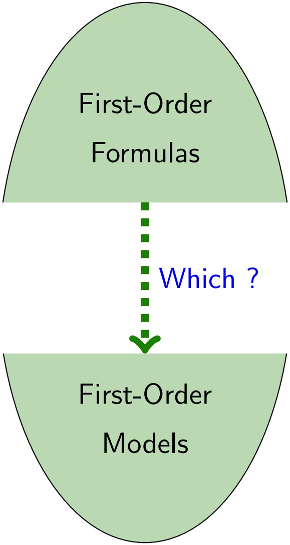


Example (At least two elements)

$\exists x;\ \exists y;\ !(x \doteq y)$

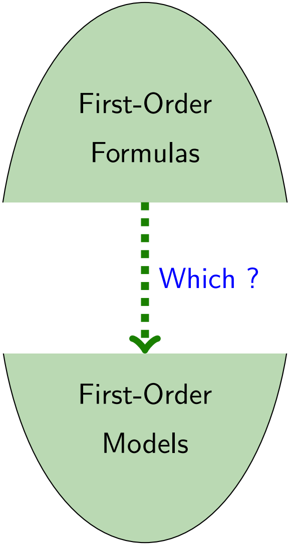
How to do this without built-in equality?

Modeling with First-Order Logic



Example (Strict partial order)

Modeling with First-Order Logic



Example (Strict partial order)

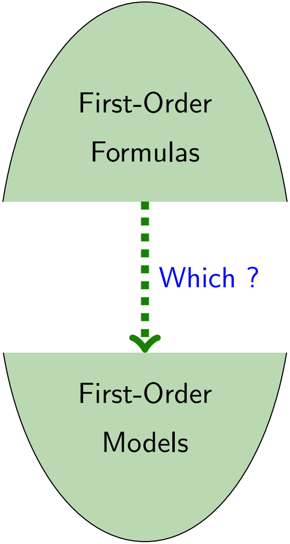
$PSym = \{< /2\}$

Irreflexivity $\forall x; !(x < x)$

Asymmetry $\forall x; \forall y; (x < y \text{ -> } !(y < x))$

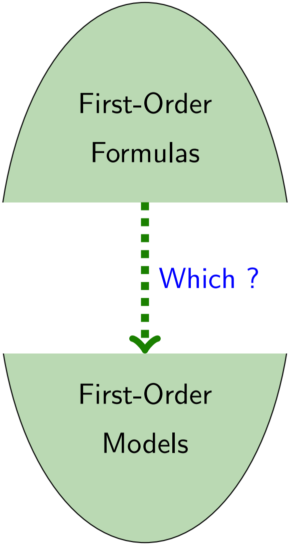
Transitivity $\forall x; \forall y; \forall z;$
$(x < y \,\&\, y < z \text{ -> } x < z)$

Modeling with First-Order Logic



Example (All models have infinite domain)

Modeling with First-Order Logic

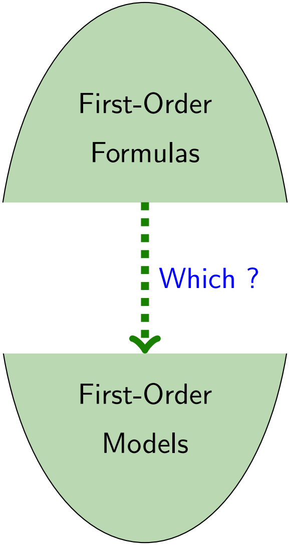


Example (All models have infinite domain)

Signature and axioms of irreflexive order plus

Existence Successor $\forall x;\, \exists y;\, x < y$

Modeling with First-Order Logic



Example (Abstract data types)

$$\text{FSym} = \{\ \text{Stack push}(\mathbf{int}, \text{Stack});$$
$$\mathbf{int}\ \text{pop}(\text{Stack});$$
$$\text{Stack nil};\ \}$$

$$\forall\, \mathbf{int}\ i;\ \forall\, \text{Stack}\ s;\ \text{pop}(\text{push}(i, s)) \doteq s$$

...

Why such a Complicated First-Order Semantics?

Why not take terms and properties at their "face value"?

Definition (Herbrand Model (untyped logic))

A first-order model where

- Domain $\mathcal{D}$ are all variable-free (i.e., ground) terms
- Each domain element is represented as a term
- Interpretation of function symbols is identity:
  $\mathcal{I}(f)(d_1, \dots, d_r) = f(d_1, \dots, d_r)$

Why such a Complicated First-Order Semantics?

Why not take terms and properties at their "face value"?

Definition (Herbrand Model (untyped logic))

A first-order model where

- Domain $\mathcal{D}$ are all variable-free (i.e., ground) terms
- Each domain element is represented as a term
- Interpretation of function symbols is identity: $\mathcal{I}(f)(d_1, \ldots, d_r) = f(d_1, \ldots, d_r)$

Major limitations of Herbrand models

- Too many different domain elements: $1 + 2$, $2 + 1$
- Natural to represent program locations as terms and domain elements as their values whose exact representation we don't know
- There are theoretical limitations as well

Outline

1. Formal Modeling
2. First-Order Logic
   - Signature
   - Terms
   - Formulas
3. Semantics
   - Domain
   - Model
   - Variable Assignment
   - Term Valuation
   - Formula Valuation
   - Semantic Notions
4. Untyped Logic
5. Modeling with FOL
6. **Summary**

Summary and Outlook

Summary

- First-order formulas defined over a signature of typed symbols
- Hierarchical OO type system with abstract and dynamic types
- Quantification over variables, no "free" variables in formulas
- Semantic domain like objects in a JAVA heap
- First-order model assigns semantic value to terms and formulas
- Semantic notions satisfiability and validity

Summary and Outlook

Summary

- First-order formulas defined over a signature of typed symbols
- Hierarchical OO type system with abstract and dynamic types
- Quantification over variables, no "free" variables in formulas
- Semantic domain like objects in a JAVA heap
- First-order model assigns semantic value to terms and formulas
- Semantic notions satisfiability and validity

Semantic evaluation is not feasible in practice

- Infinite (uncountable) number of first-order models
- Evaluation of quantified formula may involve infinitely many cases
- Next goal: a syntactic calculus allowing mechanical validity checking

Background Literature

KeYbook B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, vol 4334 of *LNCS*. Springer, 2006. Chapter 2: First-Order Logic

Fitting Melvin Fitting. *First-Order Logic and Automated Theorem Proving*, 2nd edn., Springer 1996

Ben-Ari Mordechai Ben-Ari. *Mathematical Logic for Computer Science*, Springer, 2003.

Huth & Ryan Michael Huth and Mark Ryan. *Logic in Computer Science*, Cambridge University Press, 2nd edn., 2004