# A Logic with Subtypes to talk about Java Objects

Martin Giese

Computational Logic Group

Johann Radon Institute for Computational and Applied Mathematics

Austrian Academy of Sciences

Linz, Austria

# Motivation

```
class Point {
    int x;
    int y;
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Point) {
            Point pt = (Point)obj;
            return (x == pt.x) && (y == pt.y);
        }
        return super.equals(obj);
    }
}
```

(Taken from Sun's JDK 1.5.0 source)

# Motivation

```
class Point {
    int x;
    int y;
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Point) {
            Point pt = (Point)obj;
            return (x == pt.x) && (y == pt.y);
        }
        return super.equals(obj);
    }
}
```

(Taken from Sun's JDK 1.5.0 source)

# Motivation (cont.)

Several possibilities:

- Model dynamic types of objects in an untyped logic

# Motivation (cont.)

Several possibilities:

- Model dynamic types of objects in an untyped logic

  ➠ seems inappropriate for strongly typed programming language

# Motivation (cont.)

Several possibilities:

- Model dynamic types of objects in an untyped logic

  ➠ seems inappropriate for strongly typed programming language

- Use a logic with types

# Motivation (cont.)

Several possibilities:

- Model dynamic types of objects in an untyped logic

  ➠ seems inappropriate for strongly typed programming language

- Use a logic with types
  - plain many-typed logic

# Motivation (cont.)

Several possibilities:

- Model dynamic types of objects in an untyped logic

  ➠ seems inappropriate for strongly typed programming language

- Use a logic with types

  - plain many-typed logic

    ➠ explicit coercions become a hassle

# Motivation (cont.)

Several possibilities:

- Model dynamic types of objects in an untyped logic

  ➠ seems inappropriate for strongly typed programming language

- Use a logic with types

  - plain many-typed logic

    ➠ explicit coercions become a hassle

  - logic with subtyping

# Logic with Types

Say ‘types’ instead of ‘sorts’, like in programming languages

Existing work:

- Cardelli, Wegner: *On Understanding Types, . . .* , 1985
- Goguen, Meseguer: *Order-Sorted Algebra I*, 1992
- Wernecke, Schmitt: *Tableau Calculus for Order-Sorted Logic*, 1990
- Weidenbach: *First-Order Tableaux with Sorts*, 1995

# Dynamic vs. Static Types

Two concepts of type in Java:

- *dynamic* type bestowed on every *object* when it is created.

  `new C(...)` $\leadsto$ object with dynamic type `C`

  Dynamic type never changes.

- *static* type of every *expression* in a Java program. Computed by compiler using declarations and typing rules. Objects of different dynamic type can be value of expression of same static type.

# Syntax

# Type Hierarchy

Assume a fixed

- finite set of *types* $\mathcal{T}$, and
- a *subtype relation* $\sqsubseteq$ on $\mathcal{T}$,

such that

- There is an *empty type* $\bot \in \mathcal{T}$ and a *universal type* $\top \in \mathcal{T}$.
- $\sqsubseteq$ is a reflexive partial order on $\mathcal{T}$.
- $\bot \sqsubseteq A \sqsubseteq \top$ for all $A \in \mathcal{T}$.
- $\mathcal{T}$ is closed under *greatest lower bounds* w.r.t. $\sqsubseteq$, written $A \sqcap B$.

# Type Hierarchy for Java

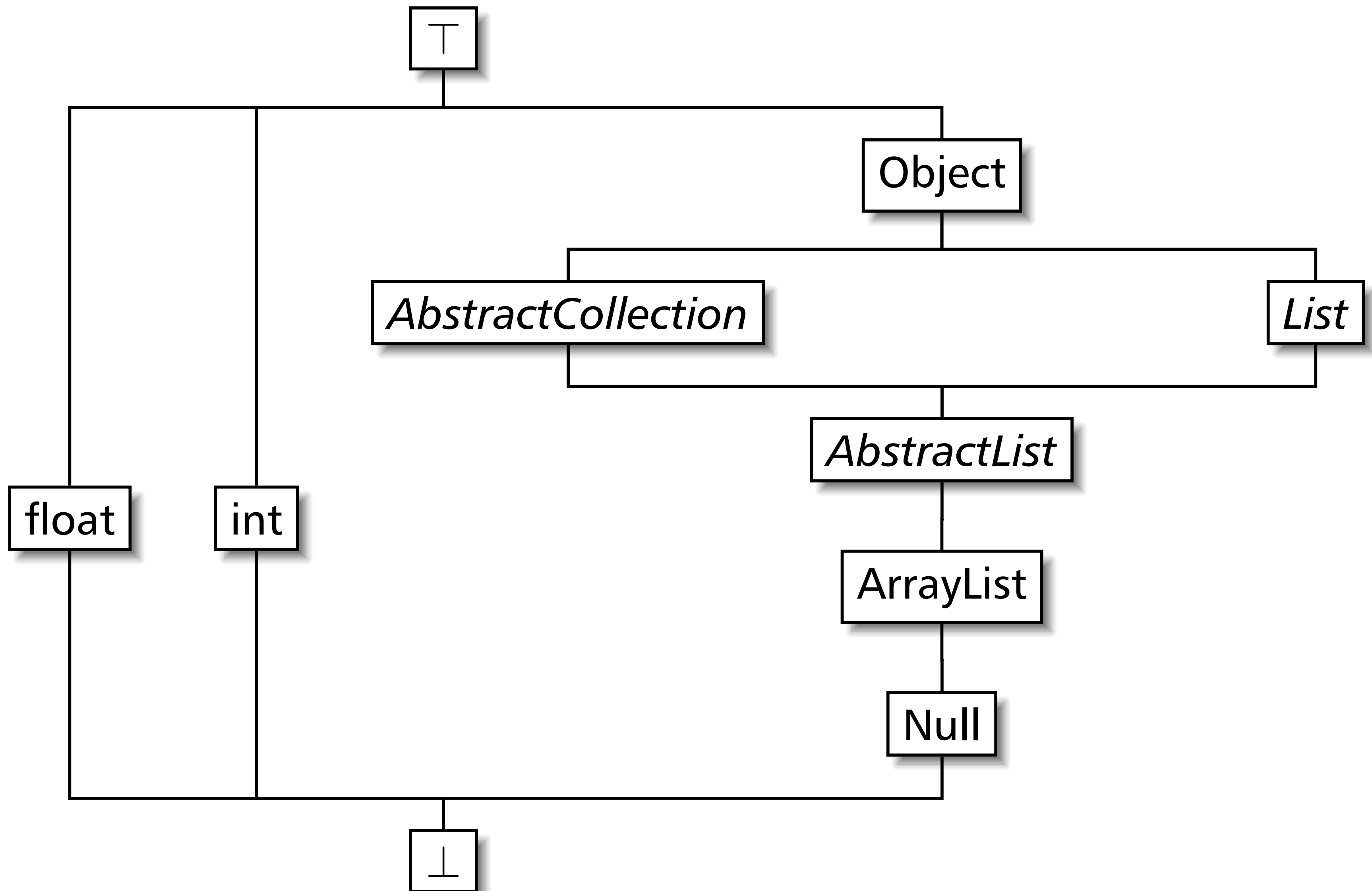
For Java, let $\mathcal{T}$ contain:

- classes, interfaces, array types from program and libraries,
- the type `Null`,
- all 'primitive types' `int`, `char`, etc.
- all intersection types required for closure w.r.t. $\sqcap$.

Define $A \sqsubseteq B$, if and only if

- $A = B$ or $A = \bot$ or $B = \top$,
- $A =$ `Null` and $B$ is a reference type, or
- There is a widening reference conversion from $A$ to $B$.

# Example Type Hierarchy



# Signatures

We assume a given set of function and predicate symbols and variables:

$$f : A_1, \ldots, A_n \to A$$

$$p : A_1, \ldots, A_n$$

$$v : A$$

No overloading: one function/predicate symbol has only one signature.

# Special Symbols

- For every type $A$, require a function symbol

$$(A) : \top \to A$$

  Write $(A)t$ instead of $(A)(t)$

- For every type $A$, require a predicate symbol

$$\sqsubseteq A : \top$$

  Write $t \sqsubseteq A$ instead of $\sqsubseteq A(t)$

- require a predicate symbol

$$\doteq : \top, \top$$

  Write $s \doteq t$ instead of $\doteq (s, t)$

# The Terms of our Logic

Inductively define the system of sets $\{T_A\}_{A \in \mathcal{T}}$ of *terms of static type $A$* to be the least system of sets such that

- $x \in T_A$ for any variable $x : A$,
- $f(t_1, \ldots, t_n) \in T_A$ for any function symbol $f : A_1, \ldots, A_n \to A$, and terms $t_i \in T_{A'_i}$ with $A'_i \sqsubseteq A_i$ for $i = 1, \ldots, n$

Write static type of $t$ as $\sigma(t) := A$ for $t \in T_A$.

# The Formulae Of Our Logic

We inductively define the set of formulae $F$ to be the least set such that

- $p(t_1, \ldots, t_n) \in F$ for any predicate symbol $p : A_1, \ldots, A_n$ and terms $t_i \in T_{A'_i}$ with $A'_i \sqsubseteq A_i$.
- $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi, \ldots \in F$ for any $\phi, \psi \in F$.
- $\forall x.\phi$, $\exists x.\phi \in F$ for any $\phi \in F$ and any variable $x$.

# Semantics

# The Domain

The semantics of a term is an element of the *domain* $\mathcal{D}$.

Each domain element $x \in \mathcal{D}$ has a *dynamic type* $\delta(x) \in \mathcal{T}$.

Going to define sematics so that:

$$\delta(\mathrm{val}_{\mathcal{S}}(\beta, t)) \sqsubseteq \sigma(t)$$

Set of admitted values for type $A$:

$$\mathcal{D}_A := \{x \in \mathcal{D} \mid \delta(x) \sqsubseteq A\}$$

Avoid complications by requiring all $\mathcal{D}_A$ for $A \neq \perp$ to be non-empty!

# Interpretation

An *interpretation* $\mathcal{I}$ assigns a meaning to every function and predicate symbol given by the signature. More precisely, for a function symbol

$$f : A_1, \ldots, A_n \rightarrow A \quad ,$$

the interpretation yields a function

$$\mathcal{I}(f) : \mathcal{D}_{A_1} \times \ldots \times \mathcal{D}_{A_n} \rightarrow \mathcal{D}_A$$

and for a predicate symbol

$$p : A_1, \ldots, A_n$$

some set of tuples of domain elements

$$\mathcal{I}(p) \subseteq \mathcal{D}_{A_1} \times \ldots \times \mathcal{D}_{A_n} \quad .$$

# Special Interpretations

For type casts:

$$\mathcal{I}((A))(x) = x \quad \text{if } \delta(x) \sqsubseteq A.$$

Otherwise: arbitrary element of $\mathcal{D}_A$.

For type predicates:

$$\mathcal{I}(\sqsubseteq A) = \mathcal{D}_A$$

For equality:

$$\mathcal{I}(\doteq) = \{(x, x) \mid x \in A\}$$

# Semantics of Terms and Formulae

A *structure* is a pair $\mathcal{S} = (\mathcal{D}, \mathcal{I})$.

A *variable assignment* is a function that assigns some domain value $\beta(x) \in \mathcal{D}_A$ to every variable $x : A$.

Term valuation

$$\mathrm{val}_{\mathcal{S}}(\beta, t)$$

and validity

$$\mathcal{S}, \beta \models \phi$$

defined as usual.

# Calculus Issues

# Wrong Cast Handling Rule

A useful tautology:

$$t \sqsubseteq A \leftrightarrow (A)t \doteq t$$

First attempt at rule for handling casts:

$$\frac{\begin{array}{c}\phi[\cdots (A)t \cdots] \\ t \sqsubseteq A\end{array}}{\phi[\cdots t \cdots]}\ \mathit{wrong}$$

$\phi[\cdots t \cdots]$ is maybe not well-typed because static type $\sigma(t)$ to general!

# The cast-add rule

Instead of removing casts, add casts where necessary:

$$\frac{\phi[\cdots t \cdots] \quad t \sqsubseteq A}{\phi[\cdots (A)t \cdots]} \text{ cast-add}$$

where $A \sqsubseteq \sigma(t)$.

Rule makes terms larger

➠ difficulties in induction of completeness proof.

# A Calculus

- There is a sound and complete sequent calculus for this logic

# A Calculus

- There is a sound and complete sequent calculus for this logic
- It’s non-trivial. FOL+equality: 7 rules, this: 14 rules

# A Calculus

- There is a sound and complete sequent calculus for this logic
- It's non-trivial. FOL+equality: 7 rules, this: 14 rules
- Completeness proof also not trivial

# A Calculus

- There is a sound and complete sequent calculus for this logic
- It’s non-trivial. FOL+equality: 7 rules, this: 14 rules
- Completeness proof also not trivial
- Usefulness in practice remains to be assessed

# Conclusion

- Defined logic with static typing, type casts, type predicates
- Motivated by Java type system
- Have a sound and complete calculus

Future Work:

- assess usefulness of logic and calculus in practice
- improve calculus for automated use

# Typing Equality

Java permits `s == t` for expressions of class type $A$ and $B$

only if one is a subclass of the other.

The JLS states that in other cases

> ‘The run-time values of the two operands would necessarily be unequal.’

# Typing Equality

Java permits `s == t` for expressions of class type $A$ and $B$

only if one is a subclass of the other.

The JLS states that in other cases

> 'The run-time values of the two operands would necessarily be unequal.'

This is wrong! Both values could be `null`.

➠ we allow equality comparisons between arbitrary types.

Model equality as a predicate symbol

$$\doteq : \top, \top$$

# The `instanceof` operator

Section 15.20.2 of JLS defines $t$ `instanceof` $A$ as follows:

> 'At run time, the result of the `instanceof` operator is true if the value of $[\ldots t \ldots]$ is not null and the reference could be cast to $[\ldots A \ldots]$ without raising a ClassCastException. Otherwise the result is false.'

# The `instanceof` operator

Section 15.20.2 of JLS defines $t$ `instanceof` $A$ as follows:

> 'At run time, the result of the `instanceof` operator is true if the value of $[\ldots t \ldots]$ is not null and the reference could be cast to $[\ldots A \ldots]$ without raising a ClassCastException. Otherwise the result is false.'

Strange definition, because `null` is member of every reference type.

Want to deviate from JLS, but don't call operator `instanceof`.