Typing OOPLs

Tobias Wrigstad
Advanced Object-Orientation
Fall/Winter 2009
DSV SU/KTH
What is a Type System?

- TAPL (Pierce, 2002, MIT Press) defines a type system as:
  
  *A tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

- Type checking is verifying and enforcing constraints of types (can happen statically and/or dynamically)

- All high-level languages performs some form of type checking

- **Type-safety:** no run-time type errors (e.g., ”no such method exception”)
What is a Type?

- A class of values, commonly
  - data type (int, float, point struct...)
  - class
  - kind (the type of a type)

- But can also be something else, like a tag (@Shared)

- Different type systems use different classifications for different purposes, regularly
  - Different kind of a assurance
  - Various optimisations
A Set-Theoretic Interpretation of Types

• Let $T$ be some type in a running program and $H$ be the set of all objects $o$ on its heap. Furthermore, let $\text{op}(T)$ be the set of operations specified by $T$.

• With these definitions, we can make an interpretation $\lbrack T \rbrack$, of $T$ s.t. $\lbrack T \rbrack$ denotes all the objects in $H$ that conform to $T$, i.e.,

$$\lbrack T \rbrack \subseteq H$$

$o \in \lbrack T \rbrack$ implies if $m$ in $\text{op}(T)$, then $o \cdot m$ exists at run-time

also, $T <: T'$ implies $\lbrack T \rbrack \subseteq \lbrack T' \rbrack$ (are subtype and subset isomorphic?)

**NB:** We could make a similar set-theoretic interpretation over some other domain, e.g., the variables in some program at compile-time.
A Set-Theoretic Interpretation of Types (cont'd)

• Thus, if a variable \( v \) has type \( T \), it means that at run-time, \( v \) will hold a reference to an object in \( \llbracket T \rrbracket \)

• **Caveat**: or null (often, why?) — *we have omitted that until now!*

• Back to set theory:

  We say that \( \llbracket T \rrbracket_{\text{null}} = \llbracket T \rrbracket \cup \{ \text{null} \} \) and \( \llbracket T \rrbracket_{\text{non-null}} = \llbracket T \rrbracket \)
Type Syntax

Class type: \( \tau = \{ f_1 : \tau_1, \ldots, f_n : \tau_n, m_1 : \rho_1, \ldots, m_m : \rho_m \} \)

Method type: \( \rho = \tau_1 \times \cdots \times \tau_n \rightarrow \tau \)
Classes-as-Types

• Defining a class extends the type system with a new type
   
   E.g.:
   
   ```java
   class Point ext. Object { int x, y; void move(int _x, _y) {...} }
   ```

   defines a type
   
   ```
   Point =
   
   \{ x: \text{int}, \ y: \text{int}, \ move : \text{int} \times \text{int} \to \text{void} \} \cup \text{Object}
   ```

• This type is notably an abstraction of all point objects

  It says nothing about behaviour

  Java’s type system not powerful enough to restrict x and y to certain ranges
Standard Type Rules (Examples from Java)

• Protect against

  Bad assignments

  Calling methods that don't exist

  Calling methods with bad arguments

  Returning bad objects from a method

  Not catching exceptions, or catching the wrong ones

  Correct operations (e.g., && and if/while on bools only)

• Essentially, types are *abstractions* and static type checking in Java protect these abstractions at compile-time by the rules above
Bad Assignments

- $\Gamma$ is a type environment, (a function from var to type)

  (Metasyntactic) variables:

  $\tau$ is a "full type" (annotation plus a class name)

  $\alpha$ is an annotation

  $T$ is a class names

  $x$ and $y$ are variable names

  $f$ is a field name

  $\Gamma(x) = T_1$  $\Gamma(y) = T_2$  $T_2 \leq T_1$

  $\Gamma \vdash x = y$

  $\Gamma(x) = T_1$  $T_1(f) = T_2$  $\Gamma(y) = T_3$  $T_3 \leq T_2$

  $\Gamma \vdash x.f = y$

  $\Gamma(x) = T_1$  $\Gamma(y) = T_2$  $T_2(f) = T_3$  $T_3 \leq T_2$

  $\Gamma \vdash x = y.f$
• Type check

\[
\begin{align*}
\text{given the rules below}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash s_1 \Gamma' & \Gamma' &\vdash s_2; \Gamma'' & \Gamma(x) = \text{int} & \Gamma(y) = \text{int} \\
\Gamma &\vdash s_1; s_2; \Gamma'' & \Gamma &\vdash \text{int} x; \Gamma \uplus \{x : \text{int}\} & \Gamma(x) = \text{int} & \Gamma(y) = \text{int} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash x + y; \Gamma
\end{align*}
\]

initially, \( \Gamma \) is \( \emptyset \). efs
• Type check

class Bar {}
class Foo { Foo self() { return this; } }
Bar b = new Foo();
Foo f = b.self();

given rules naively extrapolated from Java, and previously given rules.
Bad Method Calls

• Bad returns require us to keep track of what the current method is expected to return (can be modelled as assignment! how?)

• Exceptions require us to keep track of what exceptions can be thrown by some $m$, and what methods are called in a try-block

\[
\Gamma(x) = T_1 \quad m \in T_1 \quad \text{return}(m) = T_2 \quad \text{arguments}(m) = T_3, \ldots, T_n \\
\Gamma(y_i) = T'_i \text{ such that } T'_i \leq T_i \text{ for } y_3 \ldots y_n \\
\Gamma(z) = T'_2 \quad T_2 \leq T'_2 \\
\Gamma \vdash z = x.m(y_3, \ldots, y_n)
\]
Subtyping

• Nominal subtyping: subtyping explicitly declared by the programmer
  
  Express design intent (these concepts are related, or X defined through Y)
  
  Rigid and misses legal (wrt. types) opportunities for polymorphism

• Structural subtyping: superstructures are subtypes
  
  Do not capture design intent unless made explicit
  
  Powerful and flexible wrt. to polymorphism
Demo: subtyping and method overriding

- Covariance and contravariance

```java
class Base {
    T1 method(T2 param) {
        ...
    }
}
class Derived extends Base {
    T3 method(T4 param) {
        ...
    }
}
```

- What relations between T1 & T3, and T2 & T4 are possible?

**Danger, Will Robinson:** language Dependent.
Nominal Subtyping

- Let T1 to T4 be types defined thus:
  - T1 is defined by class Point { int x, y; }.
  - T2 is defined by class Coloured { int h; }.
  - T3 is defined by class ColouredPoint extends Point & Coloured;
  - Last, T4 is defined by class Rectangle { int x, y, h, w; }.

- Nominal subtyping:
  - T1 and T2 are not related by subtyping
  - T3 <: T1   T3 <: T2
  - T4 is not related by subtyping to T1, T2, or T4
Structural Subtyping

• Let T1 to T4 be types defined thus:
  T1 is defined by class Point { int x, y; }.
  T2 is defined by class Coloured { int h; }.
  T3 is defined by class ColouredPoint { int x, y, h; }.
  Last, T4 is defined by class Rectangle { int x, y, h, w; }.

• Structural subtyping:
  T1 and T2 are not related by subtyping

  T3 <: T1    T3 <: T2
  T4 <: T1    T4 <: T2    T4 <: T3
Parametric Polymorphism

• Adds power and flexibility to a type system without weakening type safety (has been around in ML since mid 1970’s)

• Types are defined ”with a hole” that can be filled in with other types

  Logically, we can think of filling the hole $T_1$ in a class with some type $T_2$ as creating a new class where $T_1$ has been textually replaced by $T_2$ (some languages do exactly this expansion at compile-time)

  In Java, class $T<T’>$ defines a type with a hole; Java interprets $T<T’>$ as $[T<Object>]$ unless $T’$ has a more specific bound

• Subtyping becomes somewhat trickier (look at work by Nick Cameron 2006–2009 on wildcards)
Subtyping w/ Generics: Wildcards in Java

• If $T_1 <: T_2$, naively, we could assume $T<T_1> <: T<T_2>$. However:

```java
List<T1> x = new List<T1>();
List<T2> y = x;
x.add(new T2()); T1 z = x.get(0); // BOOM!
```

• **Solution**: ? wildcard to allow subsumption in the presence of generic types

```java
List<T1> <: List<?>
```

If class `List<T>` { ... add(T) { ... } ... T get(int) { ... } ...}, then

```java
List<T1> x = new List<T1>();
List<?> y = x;
x.add(new T2()); // Does not compile
Object z = y.get(0); // OK, since everything is <: Object
```
Polymorphism and Subtyping

• **Nominal**: can be used by Point and ColouredPoint

• **Structural**: can be used by Point, ColouredPoint and Rectangle

```java
void plot(Point[] points) {
    for (Point p: points) {
        grid[p.x][p.y] = 1;
    }
}
```
Parametric Polymorphism

• Compiler can resolve "typing equations"

• Allows preserving type information, flexible code, and type-safety

```java
<A extends Point> A jxt(A p) {
    int _ = p.x; p.x = p.y; p.y = _;
    return p;
}

Rectangle r;
jxt(r).w = 100; // OK
```
Union Types and Type Constructors

• Creating types from e.g., other types (example below modified from Scala)

```java
void plot(Point with Coloured p) {
    grid[p.x][p.y] = p.h;
}
```

• Algebraic data types in O’Caml:

```ocaml
type graph = Node of string list * graph array | Nothing;;
```
Type Inference

- Static typing does not necessitate declarations in code
- Types can be inferred from operations and literals (ex. from O’Caml):

```ocaml
let rec simpleWordlistFinder graph number =
  match graph with
  | Nothing -> []
  | Node(words, children) ->
    match number with
    | [] -> words
    | n :: ns -> simpleWordlistFinder children.(n) ns
```

- May require operations to be named specially (as in e.g., O’Caml)
- Can be brittle
- Local type inference is safe, and is coming on strong
Non-Null Types

- A non-null type is a type for which the caveat on the previous slide does not apply

  *Consequence*: no null pointers.

  Study of 700 KLOC Java indicates 75% off all variables should use non-null types (Chalin & James, ECOOP 2007)
Case-Study: Adding Non-Null Types to Java

• We use @NonNull as a type modifier (à la JSR308). Variables of types annotated @NonNull denote objects in $[T]^{\text{non-null}}$. Otherwise, in $[T]^{\text{null}}$.

• Notably: $[T]^{\text{non-null}}$ subset $[T]^{\text{null}}$, so @NonNull $T <: T$

• The rest of the type system follows immediately from subtyping

• Cannot allow cast to modify @NonNull without modifying cast rules for Java
Case-Study: Loci's Type System (Take 1)

- Remember, @Thread and @Shared modifiers on types (simplification).
- For all variable types $T$, $[@Thread T] \cap [@Shared T] = \emptyset$.
- Thus, no subtyping relationship between @Thread $T$ and @Shared $T$.
- Since @Thread $T \prec @Shared T$ and @Shared $T \prec @Thread T$, standard type rules prevent shared objects from being stored in @Thread fields, and vice versa.
Case-Study: Loci's Type System (Take 1)

- Variables @Thread, @Shared or @Context (default annotation)

- Again, for all variable types T, $[@Thread T] \cap [@Shared T] = \emptyset$.

- However, it is possible that a @Context variable points to either kind of object at run-time.

- Let's investigate!

  $is [@Context T] = [@Thread T] \cup [@Shared T]$?  (Union types)
Case-Study: Loci's Type System (Take 1)

• Variables @Thread, @Shared or @Context (default annotation)

• Again, for all variable types T, \([@Thread T] \cap [@Shared T] = \emptyset\).

• However, it is possible that a @Context variable points to either kind of object at run-time.

• Let's investigate!

\textbf{NO} \hspace{1cm} \text{is} \hspace{0.5cm} [@Context T] = [@Thread T] \cup [@Shared T]? \hspace{0.5cm} \text{(Union types)}
Case-Study: Loci's Type System (Take 2)

- Variables @Thread, @Shared or @Context (default annotation)
- Again, for all variable types $T$, $\lfloor @Thread \ T \rfloor \cap \lfloor @Shared \ T \rfloor = \emptyset$.
- However, it is possible that a @Context variable points to either kind of object at run-time.
- Let's investigate!

**NO** is $\lfloor @Context \ T \rfloor = \lfloor @Thread \ T \rfloor \cup \lfloor @Shared \ T \rfloor$? (Union types)

**Realisation:** @Context does not correspond to a class of objects at run-time

At compile time, $\lfloor @Context \ T \rfloor \cap \lfloor @Shared \ T \rfloor = \emptyset$ and $\lfloor @Context \ T \rfloor \cap \lfloor @Thread \ T \rfloor = \emptyset$.

**Remember:** @Context means "same as the enclosing object"
Case-Study: Loci's Type System  (Take 2)

• @Context can be eliminated if accessed *through* a @Thread or @Shared

  \(\Gamma\) is a type environment

  (a function from var to type)

  (Metasyntactic) variables:

  \(\tau\) is a "full type" (annotation

  plus a class name)

  \(\alpha\) is an annotation

  \(T\) is a class names

  \(x\) and \(y\) are variable names

  \(f\) is a field name

\[
\begin{align*}
\Gamma(x) &= @Shared T_1 & T_1(f) &= @Context T_2 \\
\Gamma \vdash x.f : @Shared T_2 \\
\Gamma(x) &= @Thread T_1 & T_1(f) &= @Context T_2 \\
\Gamma \vdash x.f : @Thread T_2 \\
\Gamma(x) &= @Context T_1 & T_1(f) &= @Context T_2 \\
\Gamma \vdash x.f : @Context T_2 \\
\Gamma(x) &= \tau_1 & \tau_1(f) &= \tau_2 \\
\Gamma \vdash x.f : \tau_2
\end{align*}
\]

• Since @Context assignments are always from the same context they are safe, even thought we might not know if the object is @Thread or @Shared

onsdag den 11 november 2009
Case-Study: Loci's Type System (Take 2)

• Assignment rules

\[
\begin{align*}
\Gamma(x) &= \tau_1 \\
\Gamma(y) &= \tau_2 \\
\tau_2 &\leq \tau_1 \\
\Gamma \vdash x = y
\end{align*}
\]

\[
\begin{align*}
\Gamma(x) &= \alpha_1 T_1 \\
T_1(f) &= \alpha_2 T_2 \\
\Gamma(y) &= \tau_1 \\
\tau_1 &\leq (\alpha_1 \oplus \alpha_2) T_2 \\
\Gamma \vdash x.f = y
\end{align*}
\]

\[
\begin{align*}
\Gamma(x) &= \tau_1 \\
\Gamma(y) &= \alpha_1 T_1 \\
T_1(f) &= \alpha_2 T_2 \\
(\alpha_1 \oplus \alpha_2) T_2 &\leq \tau_1 \\
\Gamma \vdash x = y.f
\end{align*}
\]

<table>
<thead>
<tr>
<th>(\alpha_1 \oplus \alpha_2)</th>
<th>(\alpha_1)</th>
<th>(\alpha_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Shared</td>
<td>@Shared</td>
<td>@Shared, @Context</td>
</tr>
<tr>
<td>@Thread</td>
<td>@Thread</td>
<td>@Thread, @Context</td>
</tr>
<tr>
<td>@Context</td>
<td>@Context</td>
<td>@Context</td>
</tr>
</tbody>
</table>
Dependent Types

- Dependent type is a type that depends on a value
  
  E.g., ”the class of all arrays that are longer than the int stored in len”

- Type checking dependent programs is undecidable in the general case (if type equivalence requires running the program to compute a value)

- Allows expressing proofs in program code (compiling is verifying the proof)

- Example:

  ```java
  void access(Bar[] foo, int[0..foo.length] x) {
      ... foo[x].quux(27); // No need for bounds checking!
  }
  ```
Some Examples of Type Systems

- Session types (Takeuchi 1994)
  Capture protocols for exchanges over the wire in a type
- Type states (e.g., Deline & Fähndrich, 2003)
  Encode a state machine in a type to track correct object usage
- Ownership types (Clarke et al., 1998)
  Capture object nesting structures in types for strong encapsulation
- Linear and Unique types (e.g., Wadler 1991, Boyland et al., 2001)
  Restrict number of references to a value/object