Object typing and subtypes

John Mitchell

Reading
Chapter 10, section 10.2.3
Chapter 11, sections 11.3.2 and 11.7
Chapter 12, section 12.4
Chapter 13, section 13.3

Subtyping and Inheritance

- Interface
  - The external view of an object
- Subtyping
  - Relation between interfaces
- Implementation
  - The internal representation of an object
- Inheritance
  - Relation between implementations

Example: Smalltalk Point class

<table>
<thead>
<tr>
<th>class name</th>
<th>Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>super class</td>
<td>Object</td>
</tr>
<tr>
<td>class var</td>
<td>pi</td>
</tr>
<tr>
<td>instance var</td>
<td>x, y</td>
</tr>
</tbody>
</table>

class messages and methods
...

instance messages and methods
...

Subclass: ColorPoint

<table>
<thead>
<tr>
<th>class name</th>
<th>ColorPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>super class</td>
<td>Point</td>
</tr>
<tr>
<td>class var</td>
<td>color</td>
</tr>
<tr>
<td>instance var</td>
<td>color</td>
</tr>
</tbody>
</table>

newX:x v Y:y v C:cv (...code...)

draw (...code...)

draw

Subtyping

- If interface A contains all of interface B, then A objects can also be used B objects.

<table>
<thead>
<tr>
<th>Point</th>
<th>Colored_point</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:y:</td>
<td>x:y:</td>
</tr>
<tr>
<td>moveDx:Dy:</td>
<td>moveDx:Dy:</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>draw</td>
<td>color draw</td>
</tr>
</tbody>
</table>

Colored_point interface contains Point
Colored_point is a subtype of Point
Implicit Object types – Smalltalk/JS

- Each object has an interface
  - Smalltalk: set of instance methods declared in class
  - Example:
    
    ```
    Point (x: y: moveDx: Dy: x: y: draw)
    ColorPoint (x: y: moveDx: Dy: x: y: color: color: draw)
    ``
    - This is a form of type
      - Names of methods, does not include type/protocol of arguments

- Object expression and type
  - Send message to object
    
    ```
    p draw
    q color
    q moveDx: 5 Dy: 2
    ``
    - Expression OK if message is in interface

Subtyping

- Relation between interfaces
  - Suppose expression makes sense
    
    ```
    p msg: p pars
    ``
    - Replace p by q if interface of q contains interface of p
- Subtyping
  - If interface is superset, then a subtype
    - Example: ColorPoint subtype of Point
    - Sometimes called “conformance”
      - Can extend to more detailed interfaces that include types of parameters

Subtyping and Inheritance

- Smalltalk/JavaScript subtyping is implicit
  - Not a part of the programming language
  - Important aspect of how systems are built
- Inheritance is explicit
  - Used to implement systems
  - No forced relationship to subtyping

Smalltalk Collection Hierarchy

```
at:
Indexed
Collection
isEmpty, size, includes: ...

Set

Add
remove:

Array
replaceFrom: to: with:

Sorted collection

Dictionary
associationAt:

Subtyping

Inheritance
```

C++ Subtyping

- Subtyping in principle
  - A < B if every A object can be used without type error whenever a B object is required
  - Example:
    ```
    Point:
    int getX();
    void move(int);
    ColorPoint:
    int getX();
    int getColor();
    void move(int);
    void darken(int tint);
    ```
    - Public members
    - This is weaker than necessary Why?

Implementation of subtyping

- No-op
  - Dynamically-typed languages
    - C++ object representations (single-inheritance only)
      ```
      circle *c = new Circle(p, r);
      shape *s = c;
      ``
    - Convert
      ```
      C++ object representations w/multiple-inheritance
      C *pc = new C;
      B *pb = pc;
      A *pa = pc;
      ```
      - May point to different position in object
Smalltalk/JavaScript Representation

This is a schematic diagram meant to illustrate the main idea. Actual implementations may differ.

C++ Run-time representation

C++: virtual function lookup

C++: virtual function lookup, part 2

C++ Multiple Inheritance

Independent classes not subtypes

• Offset δ in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb
• Call to pc->g can proceed through C-as-B vtbl

• C++ does not treat ColorPoint <: Point as written
  — Need public inheritance ColorPoint : public Point
  — Why??
Why C++ design?

- Client code depends only on public interface
  - In principle, if ColorPoint interface contains Point interface, then any client could use ColorPoint in place of Point.
  - However, offset in virtual function table may differ.
  - Loses implementation efficiency (like Smalltalk).
- Without link to inheritance
  - Subtyping leads to loss of implementation efficiency.
- Also encapsulation issue:
  - Subtyping based on inheritance is preserved under modifications to base class.

Recurring subtype issue: downcast

- The Simula type of an object is its class.
- Simula downcasts are checked at run-time.
- Example:
  ```
  class A(...);
  ...
  A a = b; /* OK since B is subclass of A */
  ...
  b = a; /* Compiles, but run-time test */
  ```

Function subtyping

- Subtyping principle
  - A <: B if an A expression can be safely used in any context where a B expression is required.
- Subtyping for function results
  - If A <: B, then C -> A <: C -> B.
- Subtyping for function arguments
  - If A <: B, then B -> C <: A -> C.
- Terminology
  - Covariance: A <: B implies F(A) <: F(B).
  - Contravariance: A <: B implies F(B) <: F(A).

Examples

- If circle <: shape, then
  ```
  circle → shape
  circle → circle
  shape → shape
  shape → circle
  ```

Subtyping with functions

- In principle: ColorPoint <: Point.
- In practice: This is covariant case; contravariance is another story.

Subtyping principles (recap)

- “Width” subtyping for object types
  ```
  [m_1:p_1, ..., m_i:p_i] <: [m_1:p_1, ..., m_j:p_j]
  ```
  ```
  j
  ```

- “Depth” subtyping for object types
  ```
  [m_1:p_1, ..., m_i:p_i] <: [m_1:p_1, ..., m_j:p_j]
  ```

- Function subtyping
  ```
  σ → π <: σ' → π'
  ```
  ```
  σ → π <: σ' → π'
  ```
Subtyping recursive types

• Principle
  \[ s < t \implies \alpha(s) < \pi(t) \]
  \[ \text{s not in n(t)} \]
  \[ \text{t not in o(s)} \]

• Example
  \[ cp <: p \implies \{ \ldots \text{mv}: \text{int} \to cp \} <: \{ \ldots \text{mv}: \text{int} \to p \} \]
  \[ \text{type} \ cp = \{ \ldots \text{mv}: \text{int} \to cp \} <: \text{type} \ p = \{ \ldots \text{mv}: \text{int} \to p \} \]

Java Types

• Two general kinds of types
  – Primitive types – not objects
    – Integers, Booleans, etc
  – Reference types
    – Classes, interfaces, arrays
  – No syntax distinguishing Object * from Object

• Static type checking
  – Every expression has type, determined from its parts
  – Some auto conversions, many casts are checked at run time
    – Example, assuming \( A <: B \)
      – If \( A \ x \), then can use \( x \) as argument to method that requires \( B \)
      – Can cast \( x \) to \( A \)
      – Downcast checked at run-time, may raise exception

Classification of Java types

Reference Types
  – Object
    – Object[]
  – Throwable
  – Shape
    – Shape[]
  – Exception types
  – Exception
  – Exception[]

Primitive Types
  – boolean
  – int
  – byte
  – float
  – long

Subtyping

• Primitive types
  – Conversions: int -> long, double -> long, ...
• Class subtyping similar to C++
  – Subclass produces subtype
  – Single inheritance => subclasses form tree
• Interfaces
  – Completely abstract classes
    – no implementation
    – Multiple subtyping
  – Interface can have multiple subtypes (implements, extends)
• Arrays
  – Covariant subtyping – not consistent with semantic principles

Java class subtyping

• Signature Conformance
  – Subclass method signatures must conform to superclass
• Three ways signature could vary
  – Argument types
  – Return type
  – Exceptions
    – How much conformance is needed in principle?
• Java rule
  – Java 1.1: Arguments and returns must have identical types, may remove exceptions
  – Java 1.5: covariant return type specialization

Interface subtyping: example

```java
interface Shape {
    public float center();
    public void rotate(float degrees);
}
```
```java
interface Drawable {
    public void setColor(Color c);
    public void draw();
}
```
```java
class Circle implements Shape, Drawable {
    // does not inherit any implementation
    // but must define Shape, Drawable methods
}
```

Q: can interfaces be recursive?
Properties of interfaces

- Flexibility
  - Allows subtype graph instead of tree
  - Avoids problems with multiple inheritance of implementations (remember C++ "diamond")
- Cost
  - Offset in method lookup table not known at compile
  - Different bytecodes for method lookup
    - one when class is known
    - one when only interface is known
      - search for location of method
      - cache for use next time this call is made (from this line)

Array types

- Automatically defined
  - Array type T[] exists for each class, interface type T
  - Cannot extend array types (array types are final)
  - Multi-dimensional arrays are arrays of arrays: T[][]
- Treated as reference type
  - An array variable is a pointer to an array, can be null
  - Example: Circle[] x = new Circle[array_size]
  - Anonymous array expression: new int[][] [1,2,3,...10]
- Every array type is a subtype of Object[], Object
  - Length of array is not part of its static type

Array subtyping

- Covariance
  - if S <: T then S[] <: T[]
- Standard type error
  class A{...}
  class B extends A{...}
  B[] bArray = new B[10]
  A[] aArray = bArray  //considered OK since B[] <: A[]
aArray[0] = new A()  //compiles, but run-time error
  //raises ArrayStoreException

Covariance problem again ...

- Simula problem
  - if A <: B, then A ref <: B ref
  - Needed run-time test to prevent bad assignment
  - Covariance for assignable cells is not right in principle
- Explanation
  - interface of "T reference cell" is
    put: T -> T ref  //replaces obsolete T in S[] with T[]
    get: T ref -> T
  - Remember covariance/contravariance of functions

Afterthought on Java arrays

Date: Fri, 09 Oct 1998 09:41:05 -0600
From: bill joz
Subject: [discussion about java genericity]

actually, java array covariance was done for less noble reasons ... it made some
generic "copy" (memory copy) and like operations much easier to write...
I proposed to take this out in 95, but it was too late (...).
I think it is unfortunate that it wasn’t taken out...
it would have made adding genericity later much cleaner, and [array
covariance] doesn’t pay for its complexity today.

Java Exceptions

- Similar basic functionality to other languages
  - Constructs to throw and catch exceptions
  - Dynamic scoping of handler
- Some differences
  - An exception is an object from an exception class
  - Subtyping between exception classes
    - Use subtyping to match type of exception or pass it on ...
    - Similar functionality to ML pattern matching in handler
  - Type of method includes exceptions it can throw
    - Actually, only subclasses of Exception (see next slide)
**Exception Classes**

![Diagram of Exception Classes]

If a method may throw a checked exception, then exception must be in the type of the method.

**Why define new exception types?**

- Exception may contain data
  - Class Throwable includes a string field so that cause of exception can be described
  - Pass other data by declaring additional fields or methods
- Subtype hierarchy used to catch exceptions
  - catch <exception-type> <identifier> { ... } will catch any exception from any subtype of exception-type and bind object to identifier

**Subtyping concepts**

- Type of an object represents its interface
- Subtyping has associated substitution principle
  - If A <: B, then A objects can be used in place of B objects
- Implicit subtyping in dynamically typed languages
  - Relation between interfaces determines substitutivity
- Explicit subtyping in statically typed languages
  - Type checker may recognize some subtyping
  - Issues: programming style, implementation efficiency
- Covariance and contravariance
  - Function argument types reverse order
  - Problems with Java array covariance

**Principles**

- Object “width” subtyping
- Function covariance, contravariance
- Object type “depth” subtyping
- Subtyping recursive types

**Applications of principles**

- Dynamically typed languages
  - If A <: B in principle, then can use A objects in place of B objects
- C++
  - Class subtyping only when public base class
  - Compiler allows width subtyping, covariant depth subtyping. (Think about why...)
- Java
  - Class subtyping only when declared using “extends”
  - Class and interface subtyping when declared
  - Compiler allows width subtyping, covariant depth subtyping
  - Additional typing issues related to generics