CS 420/520 — Winter 2007

Subtyping and Substitutability or: How to be a poser



Type checking and Program Execution

Coherence:

- if
 - a program is statically analyzed and all operations assigned types...
- then
 - when that program runs, the values computed by the program and provided (dynamically) to each operation should be consistent with the results of the type assignment.



What is the nature of this consistency?

- definition time: C implements R, $C \cdot m$: $S \rightarrow T$
- message send time: $\frac{\text{o: R, v: S}}{\text{o.}m\text{ (v): T}}$
- What values can v have at run time?
 - In other words ...

v is a variable that may refer to many values. What restrictions are placed on those values by the above type assignment?



What is the nature of this consistency?

- definition time: C implements R, $C \cdot m: S \rightarrow T$
- message send time: $\frac{\text{o: R, v: S}}{\text{o.}m\text{ (v): T}}$
- What values can v have at run time?
 - In other words ...

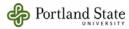
v is a variable that may refer to many values. What restrictions are placed on those values by the above type assignment?

v has exactly type S? Can we me more relaxed?



In an Object-Oriented Language:

What's the goal of the type system?



In an Object-Oriented Language:

What's the goal of the type system?

- Type systems attempt to guarantee that all messages will be understood (and that the correct method will be executed).
- So we can be more relaxed, provided that we maintain this property



Type Conformance

- If an object understands all the messages defined on a type, we say that it conforms to that type
- If an object O1 has a bigger type than object O2 then a client will not care if O1 replaces O2.
 - *bigger* ⇒ more (or the same) operations
- On the other hand, if an object has a smaller type,
 then some messages may not be understood
 - smaller ⇒ fewer (or the same) operations
- So... an instance of a bigger type can *pose* as an instance of a smaller type
 (type conformance)



Type Conformity — An example

The printerDriver takes an argument an object of type printer.



Example — continued

```
type ps
    method name () → string
    method printText(textStream, username) → status
    method printPostScript (textStream, username) → status
    method pagesPrinted → integer
end ps
```



Can the user distinguish?

The postscriptPrinter has one additional method.
 All other operations have exactly the same signature.
 If I have a some code:

```
printer p;
textStream ts;
username u;
...
p.printText(ts, u);
```

Is this OK?

- What if p is a postscriptPrinter?
- This kind of specialization supports upward compatibility.



Variance and Contravariance

The nameserver findlocalPrinter operation is initially defined:

```
nameserverV1 = type n1
method findLocalPrinter(roomNumber) →
(printer, roomNumber)
```

end n1

And is further enhanced to allow us to find a postscriptPrinter:

```
nameserverV2 = type n2

method findLocalPrinter(roomNumber) →

(postscriptPrinter, roomNumber)

end n2
```

Both nameservers support findLocalPrinter, and they take the same argument type, but they return different types.



Can I use nameserverV2 for nameserverV1?

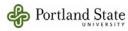
– Consider the method n:

```
Integer n (nameserverV1 ns)
{
    Integer roomNumber;
    textStream ts;
    username u;
    ...
    (printer, roomNumber) = ns.findLocalPrinter(myRm);
    printer.printTextStream(ts, u);
    return roomNumber
}
```



Enhance the print operation:

```
fileV1 = type n1
    method printOn(printer) → status
    method open() → testStream
end n1
fileV2 = type n2
    method printOn(postScriptPrinter) → status
    method open() → textStream
end n2
```



Can I use fileV2 objects in place of fileV1 objects?

```
method I (fileV1 f)
{
          printer p;
          ...
          f.printOn(p);
}
```

- If f refers to a fileV1?
- If f refers to a fileV2?



Contravariance

M.op: $A \rightarrow B$

N.op: $C \rightarrow D$

M can be used in place of N if:

C <: A

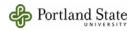
(contravariant) C has more

or the same number of operations

B <: D

(covariant) B has more or the same number of operations

- In this case M is more applicable than N (we write M <: N)
 - In any context where I have an object N, I can apply its op to any C.
 - If (instead) I have object M, since a C may replace an A, M.op can also apply to any C.



From the client's side...

M.op: $A \rightarrow B$

N.op: $C \rightarrow D$

- In any context where I expect to get a result from N.op, I expect it to be of type
- But B <: D, so a B may be used anywhere a D is...
- so it's OK for M to return a B, since that B object can pose as a D object



In Words:

Type A conforms to type B
a.k.a.(A <: B) or (A is a subtype of B)
iff:

for every operation in B there is a corresponding operation in A.
 (It is OK for A to have more operations), and

for every operation shared by A and B
 the type of the result in A conforms to the
 type of the result in B, and
 the types of arguments in B conform to the
 types of the corresponding arguments in A



Example

In Smalltalk, we can write:

```
#(1 2 3 5 7 11 13 17 19 23) collect: [: each | each * 2]
and get ...
```

· Wouldn't it be nicer to write

```
#(1 2 3 5 7 11 13 17 19 23) collect * 2
```

How can we make this work?



Polymorphism Example: The Trampoline

- #(1 2 3 4 5 6) collect * 2
 - sending the array the collect message builds a *Trampoline* with two instance variables:
 - collection: #(1 2 3 4 5 6)
 - iterationMessage: a Message
 with selector collect: and arguments: #(nil)
- When the Trampoline receives * 2
 - it replaces the *nil* in that arguments array with ... something? ... but what?
 - sends that iterationMessage to #(1 2 3 4 5 6)



From the Array's side...

- It gets the message collect: with an array containing a single argument.
- What does it do?
 - it names the argument aBlock
 - sends that aBlock the *value*: message six times, with
 1, 2, 3, 4, 5 and 6 as arguments.
 - collects the results and puts them into a new Array
 - answers that Array.



Collection » collect:

collect: aBlock

"Evaluate aBlock with each of the my elements as the argument. Collect the resulting values into a collection like me. Answer the new collection."

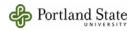
```
| newCollection |
newCollection ← self species new.
self do: [:each | newCollection add: (aBlock value: each)].
↑ newCollection
```

Does aBlock really have to be a Block?



What is the constraint on the argument of *collect:* ?

- Not that the argument is a Block
 - although blocks work just fine
- the argument must understand value:
 - and value: should expect one of the array elements as its argument.



• If Smalltalk had type declarations, *collect:* would have a signature like

```
Collection »collect: f ∈ BinaryFunction , or

Collection<e> »collect: f ∈ BinaryFunction <e>
where BinaryFunction<e> =
    type bf
    method value: <e> →<e>
end bf
```

- Blocks like [: x | x * 2] do indeed have type BinaryFunction <Integer>
- Does the Trampoline use a Block as the argument to collect:?



Message ∈ **BinaryFunction**

No! Trampoline uses a Message

doesNotUnderstand: aMessage

```
I args I args ← iterationMessage arguments. args at:args size put:aMessage. 
↑iterationMessage sentTo:collection.
```

- Is this ok?
 - why?

message »value: receiver

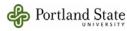
†receiver perform: selector withArguments: args



So What?

This is an example of

- Interfaces:
 - Block and Message both implement the BinaryFunction interface
- Subtyping
 - Block and Message are both subtypes of BinaryFunction
- Conformance
 - Block and Message both conform to BinaryFunction



Polymorphism?

- Some people say yes, because collect: takes arguments of different classes
- Other people say no, because collect: doesn't do anything with those arguments
 - on the contrary, it *relies* on them all having the *same type*: *BinaryFunction*.

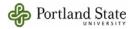


Where's the Polymorphism?

String >> valueOf: anObject

anObject ifNil: [↑ 'nil']. ↑ anObject printString

- String>>valueOf: shows Polymorphism
 - objects of varying types are provided as parameters
 - the single method does the same thing with all of those parameters
- anObject printString and anObject ifNil:
 - polymorphism or genericity?



More Subtyping Examples

if

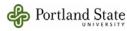
cheeseType <: foodType</pre>

then

cheeseSandwich <: sandwich

Suppose we have

function f: sandwich \rightarrow integer function g: cheeseSandwich \rightarrow integer, can f be used in place of g? Or g in place of f?



ANSWER!

We can use *f* in place of *g* because *cheesesandwich* <: *sandwich* and the function subtyping rule is contravariant for the argument.

```
c: cheeseSandwich;
```

s: sandwich

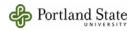
```
f(s); \leftarrow cannot use g here
```

g(s); \leftarrow **not** OK, since s cannot pose as a cheese

g(c); \leftarrow can use f here OR g

f(c):

we can replace this instance of g with f, since a cheeseSandwich can pose as a sandwich (f <: g)



More examples:

What if

```
f: Int → Sandwich
g: Int → CheeseSandwich
cheesesandwich c;
c := g(1);
```

- If I replace g with f, then c is actually bound to a sandwich.
- Can a sandwich pose as a cheeseSandwich? (NO!)
 - Here g <: f because cheeseSandwich <: sandwich</p>



Yet more sandwiches

Updatable variable:

```
var x: T
var y: S
S <: T</pre>
```

- On the RHS (expressions), y: S can pose as a T, since S <: T.
- On the LHS (variables) y: S cannot pose as a T:

```
x := aT \leftarrow OK since x must name a T x := aS \leftarrow OK since S <: T
```

- But:

 $y := aT \leftarrow Not OK$, since aT cannot pose as an S.