LATENT METHODS

Richard O'Keefe @ CS

Outline

- Background
- The problem
- Some non-solutions
- My solution
- Does it apply elsewhere

Background

- Alan Kay invented tablet computers (the Dynabook)
- His research group at Xerox produced Smalltalk-80, the second famous OO language after Simula-67.
- Simula-67 strongly influenced early C++.
- Smalltalk-80 influenced Java, but not its syntax.

Background 2

- Smalltalk was originally implemented by compilation to byte codes, executed by microcode.
- Later systems used a Just-In-Time compiler.
- I **told** you Java was influenced by Smalltalk!
- Smalltalk Virtual machines are portable between different architectures.
- I **told** you (oh, I told you)

Background 3

- Squeak is an open source descendant of Smalltalk-80. It used not to have a JIT.
- Bryce Kampjes wanted to do a research degree here with me developing a new JIT for Squeak.
- He later built Exupery, but not here.
- While I thought this would happen, I thought, “we need a baseline”.
- How hard can it be to write a Smalltalk compiler?

Smalltalk

- Everything is an object, including 137.
- Every object has a class, including classes.
- The language is single-inheritance.
- There is no type system (except in Animorphic Smalltalk, which Sun bought and killed).
- It's dynamic: **everything** can be changed at run time.
- x become: y swaps x's and y's identities

Salvation!

- There is an ANSI standard for Smalltalk.
- None of the dynamic stuff is there.
- There is no #become: .
- The GUI isn't there either (because different Smalltalk vendors had pushed it in incompatible ways).
- If only they had proof-read it, it would have been so easy...

PROJECT

- Implement ANSI Smalltalk as a batch compiler generating ISO C.
- Make the translation straightforward, but not totally dumb. Do not do type reconstruction.
- Use the Boehm garbage collector.
- AIM: to provide a reasonable baseline for evaluating other compilers.
- Once Bryce had left,
- AIM': hey, this thing looks useful,

POLICY

- Smalltalk has good support for reflection.
- Including anObject respondsTo: aSelector.
- In both free (GNU ST, Squeak, Pharo) and commercial (ST/X, Dolphin, VisualWorks, VisualAge) Smalltalks, #respondsTo: is unreliable. Example on next slide.
- Policy: an object should never claim to implement a method unless it will work sometimes.

Example

- String>>at: index put: aCharacter
  treats a string as a mutable array.
- Symbol is a subclass of String for unique strings (Lisp atoms).
  Symbol>>at: index put: aCharacter
  ```
      self shouldNotImplement.
  ```
- So #at:put: **never** makes sense for symbols, but it's in their interface.
- I refuse to do that.

Java does it too

- The default implementation of java.lang.Iterator.remove() throws UnsupportedOperationException.
- That's a precise equivalent of Smalltalk's #shouldNotImplement.
- So in Java as in Smalltalk, **reflection is unreliable.** A method can be in an object's interface even if it can **never** be used on that object.
- I don't want to do this.

The problem: duplicate code

- Part of the Collection hierarchy:
  Collection
    AbstractKeyedCollection
      AbstractSequence
        ReadOnlyArray
          Array
        ReadOnlyString
          String
        ReadOnlyByteArray
          ByteArray
-

Mutation Methods

- #at:put: has to be different for each kind of mutable array because the generated C code has to be different.
- atAll: indices put: anElement
    indices do: [:each |
        self at: each put: anElement]
  doesn't have to be different.
- But it cannot be inherited; the common ancestor of Array, String, ByteArray, BooleanArray, BitArray, ShortArray, Float{E,D,Q}Array is not mutable
- So I need 9 copies of this method... and others!

Stream Classes

- Stream
  - InputStream
    - ReadStream "based on sequences"
  - OutputStream
    - WriteStream "based on sequences"
      - ReadWriteStream
- You can tell what's coming, can't you?

Stream Methods

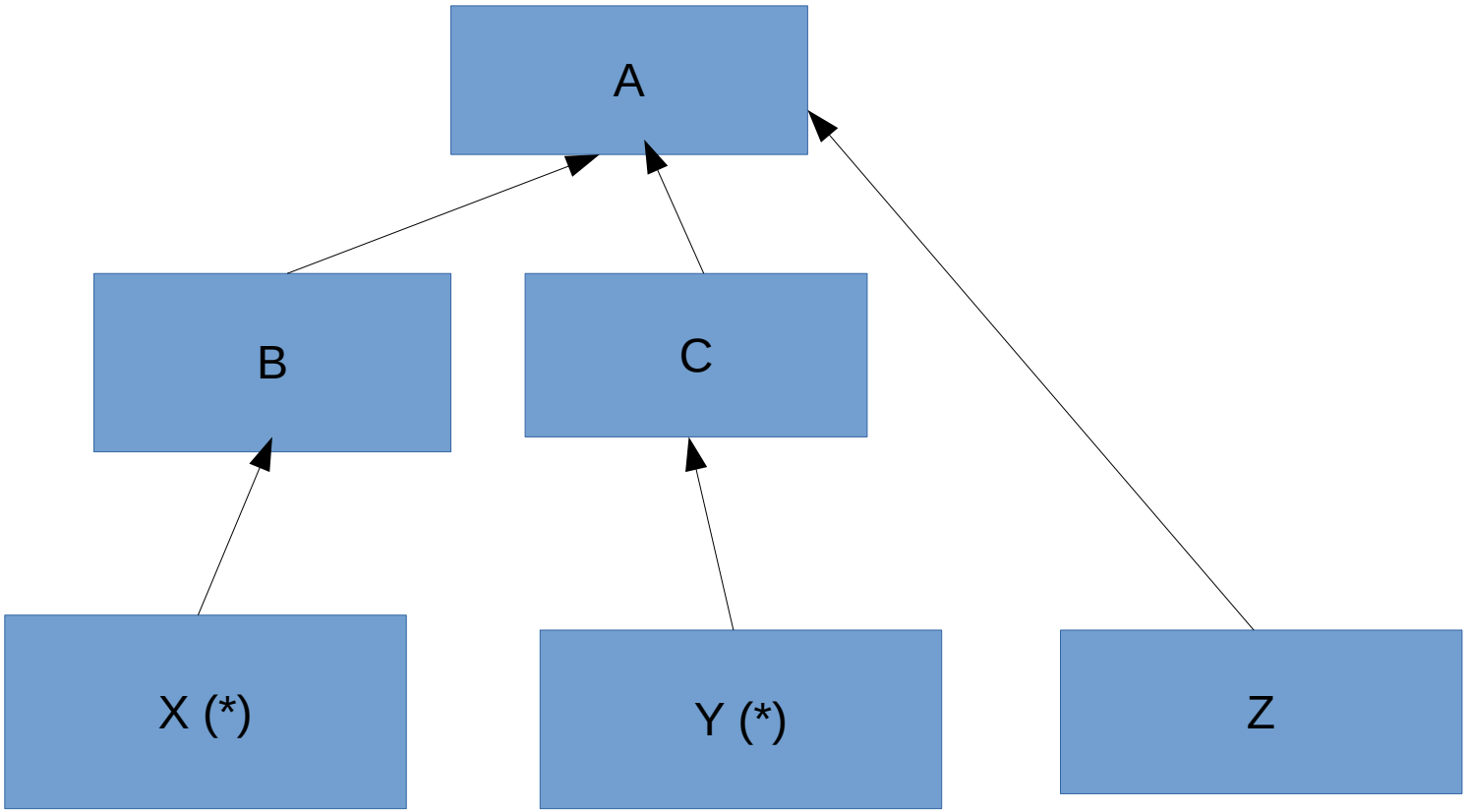
- upTo: endObject

```
|result item|
result := OrderedCollection new.
[self atEnd or: [(item := self next) = endObject]]
    whileFalse: [result addLast: item].
^result
```

- has 9 versions. Some are genuinely different. ReadStream can return a slice of its container without an OrderedCollection. Some are *not*.

The Problem

- (1) Reduce code duplication
- (2) keeping reflection honest
- (3) with no difficulty about what is inherited


A
B
C
X (*)
Y (*)
Z

Non-Solution: historic Smalltalk

- The traditional Smalltalk approach is that if a method should appear in class X and class Y you put it in A, their nearest common ancestor.
- For example, Smalltalk has mutable arrays (X) and immutable strings (Y). Their common ancestor is ArrayedCollection. The #at:put: method goes there. But Interval (lb to: ub by: step) then inherits it and has to block it.
- Code duplication is avoided (1).
- Reflection is not reliable (2).

Non-solution: Java interfaces

- Classic Java interfaces do not solve the code duplication problem but *create* it.
- Declaring that a class implements an interface creates an *obligation* to implement its methods. It does not provide code you can inherit.
- Java 8 default methods I count as multiple inheritance. They answer the question “how to add new features to an interface with existing implementors that lack them”.
- Two parent interfaces with default definitions for the same method = multiple inheritance. Oops.

Non-solutions: macros

- ArrayedCollection subclass: #Array
  $include 'array-mutation-methods.ist'

  ArrayedCollection subclass: #String
  $include 'array-mutation-methods.ist'
- Means I only have to *write* a definition once, and reflection works.
- But the compiler has to *compile* multiple copies of such definitions. Object code bloat.
  We also get obscure source code.

Non-solution: multiple inheritance

- If P has parents Q and R, and both define m, which m does p inherit?
- Common Lisp: each class has a class precedence list. Two classes with the same set of ancestors may inherit different things.
- Dylan: like Common Lisp, but computes class precedence list differently.
- Eiffel: elaborate language for connection

Multiple Inheritance 2

- C++: dislike of its complexity is why Java and C# became popular.

- Multiple inheritance solves (1) code duplication and (2) honest reflection (if you have it) but (3) it's hard to tell where things will come from.
- Multiple inheritance is **too** powerful.

Key insight

- Smalltalk is *nearly* right. We do want to define a "common" method once, and the common ancestor A is a good place to do it.
- The problem is that placing a definition in A makes it available in B C and Z as well as X and Y.
- But **definition** and **availability** don't have to be the same thing!

Latent Methods

- A latent method is **defined** in a class together with a condition.
- It becomes **available** in a (sub)class when the condition is satisfied in that class.
- But what kind of condition?

Latent method = template method

- In Design Patterns, a template method is an algorithm, typically in an abstract class, that calls methods that are overridden in subclasses so that the subclasses can customise its behaviour.
- atAll: indices put: anElement
    indices do: [:each |
        self at: each put: anElement]

Available when?

- It makes sense for a latent method to become available in precisely those (sub)classes where it is first true that all the self methods exist.
- Actually, template methods are *always* available. They should be latent ones.
- People make mistakes. So we should say which self methods we *intend* to be filled in by subclasses

Example

- Stream
  latent #(atEnd next) methods:
    do: aBlock
      [self atEnd] whileFalse: [
        aBlock value: self next].
    ...
  latent #(nextPut:) methods:
    nextPutAll: aCollection
      aCollection do: [:each | self nextPut: each].

Implementation

- In a typed single-inheritance language, latent methods go in the virtual function table just like any other methods. Availability is mainly enforced at compile time. There is no run-time overhead for ordinary dynamic dispatch.
- Reflection is trickier, but can use the same technique that my Smalltalk uses. Calls through reflection pay a small price.

Implementation 2

- SmartEiffel introduced the idea of *not* using a virtual function table. Instead an object begins with a class *number*, and dynamic dispatch uses a sort of optimised switch().
- My Smalltalk copied this. Baseline, remember? This obviously wasn't going to be fast enough. Except it was.

Dynamic dispatch

- object.selector(args...) =>
  selector'(object, args) where
  Word selector'(Word self, Word ...) {
    Tag const c = CLASSNO(self);
     /* tree of if () statements */
     return does_not_understand(
        "selector", self, ...);
  }
- branches of the tree can share (thanks, goto!)

Duplication?

- No. If multiple class number ranges correspond to a single definition, they can all be-or-jump-to a single translation. This is needed anyway to handle overriding.
- Subproblem (1) is solved.

Reflection?

- object respondsTo: #selector ==> responds(CLASSNO(object), selector'') where selector'' is an array of half-open intervals of class numbers where *some* definition of the selector is **available** and responds() does a binary search.
- The test is logarithmic in the number of *intervals,* which is usually much less than the number of *classes*.

Reflection? 2

- The where-available tables can be shared between selectors.
- Currently, 874 classes (*2 for metaclasses), 7697 selectors, 1724 unique range lists, longest has 47 ranges, average 2.75, memory to hold range lists = 56612 bytes. 860258 lines of generated C for everything else.
- The scheme is practical.

Really dynamic languages

- The dynamic dispatch technique I'm using relies on whole-program compilation to number the classes.
- Smalltalk is really dynamic.
- Implementations use inline caches, possibly polymorphic inline caches, which still work. Fallback searches the object's class and ancestors checking each's method dictionary. This is easy to adapt.