Delegation Revisited

Reuse Mechanisms in a Statically Typed, Expression-Oriented Language

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Alex Potanin
Victoria University of Wellington
alex@ecs.vuw.ac.nz

Abstract

How should code reuse be supported in a statically-typed, expression-oriented programming language? Neither function-based reuse nor class-based inheritance provides a good solution in this setting. We question the requirements for code reuse, proposing that future work evaluate whether open recursion is really needed to support cases commonly seen in industry. In addition, we explore the design space of forwarding and delegation mechanisms, finding that there are solutions that are surprisingly expressive despite their simplicity.

Function-based reuse works well at the small scale, but at the medium scale we would like to compose groups of functions and perhaps state (i.e. objects). Statically-typed inheritance-based reuse works with classes as top-level declarations, but this choice is incompatible with an expression-oriented language design. First-class classes restore expression orientation, but make static typing complex.

Wyvern is a new secure object-oriented language in sore need of a code reuse functionality. Being object-oriented aficionados we discuss different ways *delegation* can be implemented in Wyvern exploring different possibilities similar to Jones et al. exploring inheritance [2].

1. Reuse in a Statically-Typed, Expression-Oriented Setting

An expression-oriented programming language is built primarily out of expressions that reduce to first-class values. This design provides significant expressiveness benefits compared to languages that provide some abstractions only as top-level declarations. Similarly, statically typed programming languages offer compelling benefits for understanding programs at scale and for building effective tools. A language that combines these characteristics could offer outstanding benefits to programmers, but it has been difficult to develop a good design that can provide good code reuse while keeping the type system simple enough to be usable. The most notable prior design in this space required complex typing mechanisms such as row polymorphism to reason about first-class classes [8].

Functional languages make up some of the most successful expression-oriented languages to date, and good static type systems have been developed. However, while functions provide excellent reuse at the small scale, libraries and frameworks benefit greatly from medium-scale reuse abstractions that combine multiple functions into objects [1] and combine object descriptions using mechanisms such as inheritance. Dynamically typed object-oriented languages such as Smalltalk provide excellent expressiveness, but lack static typing. Statically-typed object-oriented languages, in contrast, have tended to support classes as top-level declarations, limiting expressiveness so that classes cannot be passed around as first-class objects, but keeping the type system relatively simple. A notable exception is Racket, which provides (gradual) static typing and first-class classes, at the cost of requiring more complex typing mechanisms such as row polymorphism [8].

In the Wyvern project, we are seeking to develop an expression-oriented language that supports good reuse facilities with a relatively simple, albeit carefully designed, static type system. We start with a pure object-oriented core that models objects as records with methods and fields as members, following Bruce et al. [3]. Our model considers classes, modules, and other extensions as simply additional convenient "sugar" that can be translated to a pure object-oriented core [6]. For example, in Wyvern's design a "class" is simply an object with a factory method to create "instances". To promote simplicity and expressiveness, Wyvern's subtyping relation is structural; support for nominal types is provided thorugh a separate tagging mechanism [4].

This paper explores whether it is possible to support an expressive object-level reuse mechanism in this setting, without sacrificing the simplicity of the static type system or the expression-oriented nature of the language. To remain expression-oriented, we seek to avoid tying our reuse mech-

```
class AbstractCollection {
    public abstract int size();
    public int isEmpty() {
        return size() == 0;
    }
}
```

Figure 1: java.util.AbstractCollection code excerpt

anism to statically-declared classes, and so like Jones *et al.* we seek something close in expressiveness to object inheritance without classes [2]. Considering mechanisms such as delegation is inspired by our initial investigation showing that delegation-based formalism captures type state in a simpler and richer fashion than a class-based formalism [5].

The next section explores whether we really need a mechanism for reuse that includes open recursion, the source of much of the complexity in the semantics of inheritance. We follow with a section that considers a variety of other mechanisms that could support open recursion in the new patterns of building open recursion out of simpler primitives in the (hopefully rare) cases when it really is needed.

2. The Need for Open Recursion

As open recursion is the source of considerable complexity when reasoning about inheritance and related delegation mechanisms [5], we can hope to keep our type systems (and reasoning systems built on top of them) simpler if the language does not build open recursion into its reuse mechanism.

How important is it for a reuse mechanism to support Open Recursion? A recent empirical study found that the median program in a Java code corpus used downcalls in 34% of relationships between classes – and downcalls indicate a use of open recursion [10]. This data initially makes it seem that open recursion is commonly needed.

However, the study did not consider whether designs that use downcalls could be written without them. We will show an example of how the open recursion in `java.util.AbstractCollection` could be replaced by simple forwarding by separating core functionality from overrides in the forwarding chain.

Consider the design of `java.util.AbstractCollection` in Figure 1. Here the `isEmpty()` method includes a downcall to the `size()` method. However, this design could perhaps be refactored into a design that uses forwarding, without open recursion. The `AbstractCollection` constructor could take a `BasicCollection` argument that implements `size()` and other basic methods in terms of the appropriate data structures, and `AbstractCollection` would forward calls to `size()` down to the underlying collection. If a programmer wants to override methods such as `isEmpty()`, that can be done by in turn wrapping the `AbstractCollection`. This design has the drawback of possibly requiring two objects to support both the basic functionality and the overrides, but it

```
module abstractCollection

resource type BasicCollection
    def size():Int
    // other methods....

resource type Collection extends BasicCollection
    def isEmpty():Bool
    // other methods...

def make(delegator: BasicCollection):Collection
    new
        forward BasicCollection to delegator
        def isEmpty():Bool = (delegator.size() == 0)
```

Figure 2: An AbstractCollection in Wyvern

gains considerable simplicity by not requiring open recursion. One such example is shown in Wyvern in Figure 2.

We hypothesize that a language design that provides a forwarding composition mechanism, without the open recursion in inheritance and delegation, could support many of the designs identified by Tempero *et al.*, and we hope that future empirical studies could evaluating this hypothesis.

3. Exploring Delegation in Wyvern

Our first proposal for delegation support in Wyvern builds a little syntactic sugar on simple function forwarding to express most inheritance patterns. In Figure 3, the type of `Animal` describes instances that can `eat` and `swallow` while a `Dog` can also `bark`. When we construct an instance of an `Animal` we distinguish between a current instance being created (`animalSelf`) and the instance that we might be extending (`childSelf`) that will be given to us as a parameter to a method that is delegated to the `Animal`. Thus, an animal can have a local version of eating and a dog that happens to be an animal can have its eating *delegated* to the animal instance.

The support for such downcalls is provided by the `delegate T to x` construct which looks at whether each method `m` in `T` is defined with the same signature in the current new statement (in which case it does nothing) or if it is defined in the type of `x` (in which case a wrapper is generated that directly calls `x.m`) or finally if it is defined in the type of `x` with the same signature and an extra self parameter (in which case a wrapper calls `x.m(..., this)`).

We believe that making delegation explicit is better than implicitly allowing the superclasses to interact with subclasses using downcalls (or breaking "uniform identity" as defined by Jones et al. [2]) which our preliminary investigation of the Qualitas Corpus [9] shows to be occurring in 8% of the *extended* classes and is a potential source of hidden bugs and unintended functionality. In general, inheritance introduces implicit rules about code execution that are not obvious to the programmer, and thus we argue it *increases* complexity.

```
type Animal
  def eat()
  def swallow()

type AnimalDelegate
  def eat(self:Animal)
  def swallow()

type Dog extends Animal
  def bark():String

def makeAnimal():AnimalDelegate = new animalSelf =>
  def eat(@Self childSelf:Animal)
    childSelf.swallow()
    animalSelf.swallow()
  def eat() = new LocalStuffToDo()
  def swallow() = new Unit()

def makeDog(base:AnimalDelegate):Dog = new
  delegate Animal to base
  // the ?delegate? declaration above is equiv. to:
  // def eat() = base.eat(this)
  def bark():String = ''Woof''
  def swallow():Unit = new Unit()

val base:AnimalDelegate = makeAnimal()
val a:Animal = makeDog(base)
a.eat() // Dog.swallow(), then Animal.swallow()
a.bark() // ''Woof''
```

Figure 3: Example of Delegation in Wyvern

4. Where To Next?

Our Wyvern proposal above is just the first of the ones we intend to discuss in the presentation. Although it solves the technical problem, there is implementation complexity showing in the form of an extra type (`AnimalDelegate`) and an extra explicit `self` parameter (one for the current object and one for the original object which are of different types!). Can we do better?

One option we will discuss is *"The Dylan/CLOS Solution"*: every method takes self as a parameter, but it defaults to the receiver. There is a syntax for specifying a replacement. So `x.m()` passes `x` as self, but something like `x.m(self=y)` replaces it. In a typed setting there is still some complexity because the passed-in self parameter has a different type than the (lone) parent object. A simple example is shown in Figure 4.

A second option to discuss is to provide a *Forwarding Construct* that does not pass "self". This is less expressive (forwarding, not delegation) but is semantically clean and gets rid of all the ugliness above. We intend to explore (and push) the limits of the expressiveness of forwarding in our talk.

A third option is to make the *Parent have a Pointer to the Child* (but potentially awkward due to either recursive initialisation, or reusing the same parent with multiple children). Uniqueness types or a recursive initialisation primitive (e.g. placeholder types by Servetto et al. [7]) could facilitate this, with the latter likely being a better solution to the problem. While we avoid the extra types and awkward self parameters, we will pay the space cost of explicit extra pointers to the child and it would make it harder to merge objects.

A fourth alternative is provide a *CLASS* construct that builds everything above inside it and while "under the covers" classes are still implemented in terms of delegation the programmer does not need to deal with the complexity. This solution hides as much complexity as the usual class based inheritance, but at least it can be explained (and perhaps typed) in terms of expression-oriented primitives.

```
// This is per type not per method solution as first try.
type Animal
  def eat()
  def swallow()

type Dog extends Animal
  def bark():String

// Self is assigned mine (this) by default.
def makeAnimal(self:Animal = mine):Animal = new mine =>
  def eat()
    self.swallow()
    mine.swallow()
  def swallow() = new Unit()

def makeDog(self:Animal):Dog = new mine =>
  delegate Animal to self
  // the ?delegate? declaration above is equivalent to:
  // def eat() = self.eat()
  def bark():String = ''Woof''
  def swallow():Unit = new Unit()

val a:Animal = makeAnimal()
val d:Animal = makeDog(a)
d.eat() // calls Dog.swallow(), then Animal.swallow()
d.bark() // ''Woof''
```

Figure 4: Alternative "Dylan-Style" Example of Delegation

References

[1] Jonathan Aldrich. The power of interoperability: Why objects are inevitable. In *Onward! 2013*, pages 101–116. ACM, 2013.

[2] Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object inheritance without classes. In *ECOOP*, 2016.

[3] Benjamin Pierce Kim Bruce, Luca Cardelli. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.

[4] Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *ECOOP*, pages 174–197, 2015.

[5] Du Li, Alex Potanin, and Jonathan Aldrich. Delegation vs inheritance for typestate analysis. In *FTfJP*, 2015.

[6] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *MASPEGHI '13*, pages 9–16, New York, NY, USA, 2013. ACM.

[7] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix. In *ECOOP*, volume 7920, pages 205–229. Springer Berlin Heidelberg, 2013.

[8] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Mattias Felleisen. Gradual typing for first-class classes. In *OOPSLA*, pages 793–810, 2012.

[9] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference*, 2010.

[10] Ewan Tempero, Hong Yul Yang, and James Noble. What programmers do with inheritance in java. In *ECOOP*, pages 577–601. Springer, 2013.