The Essence of Inheritance

Andrew P. Black[1], Kim B. Bruce[2], and James Noble[3]

[1] Portland State University, Oregon, USA,
`black@cs.pdx.edu`,
[2] Pomona College, Claremont, California, USA,
`kim@cs.pomona.edu`
[3] Victoria University of Wellington, New Zealand `kjx@ecs.vuw.ac.nz`

Abstract. Programming languages serve a dual purpose: to communicate programs to computers, and to communicate programs to humans. Indeed, it is this dual purpose that makes programming language design a constrained and challenging problem. Inheritance is an *essential* aspect of that second purpose: it is a tool to improve communication. Humans understand new concepts most readily by *first* looking at a number of concrete examples, and *later* abstracting over those examples. The essence of inheritance is that it mirrors this process: it provides a formal mechanism for moving from the concrete to the abstract.

Keywords: inheritance, object-oriented programming, programming languages, abstraction, program understanding

1 Introduction

Shall I be abstract or concrete?

An abstract program is more general, and thus has greater potential to be reused. However, a concrete program will usually solve the specific problem at hand more simply.

One factor that should influence my choice is the ease with which a program can be understood. Concrete programs ease understanding by making manifest the action of their subcomponents. But, sometimes a seemingly small change may require a concrete program to be extensively restructured, when judicious use of abstraction would have allowed the same change to be made simply by providing a different argument.

Or, I could use inheritance.

The essence of inheritance is that it lets us avoid the unsatisfying choice between abstract and concrete. Inheritance lets us start by writing a concrete program, and later abstracting over a concrete element. This abstraction step is *not* performed by editing the concrete program to introduce a new parameter, as would be necessary without inheritance. To the contrary: inheritance allows us to treat the concrete element *as if it were a parameter*, without actually changing the code. We call this ex post facto parameterization; we will illustrate the process with examples in Sections 2 and 3.

Inheritance has been shunned by the designers of functional languages. Certainly, it is a difficult feature to specify precisely, and to implement efficiently, because it means (at least in the most general formulations) that any apparent constant might, once inherited, become a variable. But, as Einstein is reputed to have said, in the middle of difficulty lies opportunity. The especial value of inheritance is as an aid to program understanding. It is particularly valuable where the best way to understand a complex program is to start with a simpler one and approach the complex goal in small steps.

Our emphasis on the value of inheritance as an aid to human understanding, rather than on its formal properties, is deliberate, and long overdue. Since the pioneering work of Cook and Palsberg (1989), it has been clear that, formally, inheritance is equivalent to parameterization. This has, we believe, caused designers of functional languages to regard inheritance as unimportant, unnecessary, or even undesirable, arguing (correctly) that it can be simulated using higher-order parameterization. This argument misses the point that two formally-equivalent mechanisms may behave quite differently with respect to human cognition.

It has also long been known that *Inheritance is Not Subtyping* (Cook et al., 1990). In spite of this, many programming languages conflate subtyping and inheritance; Java, for example, restricts the use of inheritance so that the inheritance hierarchy is a sub-tree of the type hierarchy. Our goal in this paper is to consider inheritance as a mechanism in its own right, quite separate from the subtyping relation. We are aided in this goal by casting our example in the Grace programming language (Black et al., 2012), which cleanly separates inheritance and subtyping.

The form and content of this paper are a homage to Wadler's "Essence of Functional Programming" (1992), which was itself inspired by Reynold's "Essence of ALGOL" (1981). The first example that we use to illustrate the value of inheritance, discussed in Section 2, is based on Wadler's interpreter for a simple language, which is successively extended to include additional features. (Reynolds (1972) had previously defined a series of interpreters in a classic paper that motivated the use of continuations.) The second example is based on Armstrong's description of the Erlang OTP Platform (Armstrong, 2007, Ch 16); this is discussed in Section 3. We conclude in Section 4, where we also discuss some of the consequences for type-checking.

2 Interpreting Inheritance

This section demonstrates the thesis that inheritance enhances understandability, by presenting several variations of an implementation of a fragment of an expression language. Wadler (1992) uses an interpreter for the lambda calculus, but we use a simpler language inspired by the "First Monad Tutorial" (Wadler, 2013).

We show our examples in Grace (Black et al., 2012), a simple object-oriented language designed for teaching, but no prior knowledge of the language is assumed. What the reader will require is a passing familiarity with the basics of object-oriented programming; for general background, see Cox (1986), or Black et al. (2009). Like Haskell, Grace does not require the programmer to specify types on every declaration, although, also like Haskell, the programmer may well find that adding type annotations makes it easier to find errors.

2.1 Variation Minus One: Object-Oriented Evaluation

We start with a simple object-oriented evaluator for Wadler's tutorial example (2013) — mathematical expressions consisting only of constants con and division div — so 1/2 is represented as div(con 1, con 2). The object-oriented style integrates the data and the descriptions of the computations on that data. Thus, there is no separation between the "interpreter" and the "terms" that it interprets; instead the object-oriented program contains "nodes" of various kinds, which represent *both* expressions *and* the rules for their evaluation.

In Grace, con and div are classes; they correspond to Wadler's algebraic data constructors. Grace classes (like those in O'Caml and Python) are essentially methods that return new objects. Evaluation of an expression in the example language is accomplished by requesting that the objects created by these classes execute their own eval methods, rather than by invoking a globally-defined function interp. The resulting representation of simpleExpressions is straightforward. In addition to the eval methods, the objects are also given asString methods that play a role similar to that of Wadler's show∗ functions: they return strings for human consumption. (Braces in a string constructor interpolate values into the string.)

module "simpleExpressions"

```
class simpleExpressions {
    class con(n) {
        method asString −> String { "con {n}" }
        method eval −> Number { n }
    }
    class div(l, r) {
        method asString −> String { "div({l}, {r})" }
        method eval −> Number { l.eval / r.eval }
    }
}
```

Given these definitions, we can instantiate an expression tree, evaluate it, and print out the result. (Grace's **def** is similar to ML and Scala's **val**).

```
import "simpleExpressions" as s
def e = s.simpleExpressions
def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)          // prints 42
def error = e.div(e.con 1, e.con 0)
print (error.eval)          // prints infinity
```

Wadler (1992) does not present code equivalent to this variation, which is why we have labelled it "Variation Minus One". Instead, he starts with "Variation Zero", which is a monadic interpreter based on a trivial monad.

2.2 What is a Monad?

For our purposes, a monad is an object that encapsulates an effectful computation. If the underlying computation produces an answer `a`, then the monad can be thought of as "wrapping" the computation of `a`. If you already understand monads, you should feel free to skip to Section 2.3.

All monad objects provide a "bind" method, which, following Haskell, we spell `>>=`. (In Grace, binary methods can be named by arbitrary sequences of operator symbols; there is no precedence.) In general, a monad will also encapsulate some machinery specialized to its particular purpose and accessible to its computations. For example, the output monad of Section 2.6 encapsulates a string and an operation by which a computation can append its output thereto.

The bind method `>>=` represents sequencing. The right-hand argument of `>>=` is a Grace code block, which denotes a λ-expression, and which represents the computation that should succeed the one in the monad. (The successor computation is like a continuation; Wadler (1992, §3) explores this relationship.) The request `m>>= { a -> ... }` asks the monad object `m` to build a new monad that, when executed, will first execute `m`'s computation, bind the answer to the parameter `a` of the block, and then execute the successor computation represented by `...`. Think of the symbol `>>=` as piping the result of the computation to its left into the function object to its right. The block should answer a monad, so the type of `>>=` is `(Fun<a, Monad<b>>) -> Monad<b>`. (The Grace type `Fun<x, y>`, defined in the standard prelude, describes a function that takes an argument of type `x` and returns a result of type `y`; the application of a function is performed explicitly using the method `apply`.) Although `Monad<a>` is parameterized by the type `a`, Grace's gradual type system allows us to omit type parameters. Omitted types and type parameters are treated as `Unknown`, which means that the programmer chose not to state the type.

In the object-oriented formulation of monads, the monad classes construct monad objects, and thus subsume the operation that Haskell calls `unitM` or **`return`**. In the sections that follow, we introduce four monad classes: `pure(contents)`, which encapsulates pure computations that have no effects; `raise(reason)`, which encapsulates computations that raise exceptions; `stateMonad(contents)`, which encapsulates stateful computations; and `monad(contents)output(str)`, which encapsulates computations that produce output. The name of this last class is an example of a method name with multiple parts, a Grace feature designed to improve readability that is similar to Smalltalk method selectors. The class `monad()output()` expects two argument lists, each with a single argument.

Informally, the monad classes get us into a monad, and `>>=` gets us around the monad. How do we get out of the monad? In general, such operations require a more *ad hoc* design. In this article, it will suffice to provide the monads with `asString` methods, which answer strings.

2.3 Variation Zero: Monadic Evaluation

We now *adapt* the code of Section 2.1 to be monadic. Here we see the key difference between an approach that uses inheritance and one that does not: rather than define the new expression evaluation mechanism from scratch, we make a new monadicExpression module that uses inheritance to modify the components imported from the previous simpleExpressions module.

For a monadic interpreter, we of course need a monad. In Grace, the parameterized *type* Monad<a> characterizes the common interface of all monads. The only concrete monads in this module are created by a *class* called pure, so-called beacuse it encapsulates effect-free computations that answer pure values. The bind method >>= applies its parameter k to the contents of the monad.

module "monadicExpressions"

```
   import "simpleExpressions" as se
 2
   type Monad<a> = type {
 4     >>= <b> (k:Fun<a, Monad<b>>) −> Monad<b>
   }
 6 class monadicExpressions {
       inherits se.simpleExpressions
 8
       class pure<a>(contents:a) −> Monad<a> {
10         method asString −> String { "pure({contents})" }
           method >>= <b> (k:Fun<a, Monad<b>>) −> Monad<b> {
12             k.apply(contents)
           }
14     }

16     class con(n) {
           inherits se.simpleExpressions.con(n)
18             alias simpleEval = eval
           method eval −> Monad<Number> is override { pure(simpleEval) }
20     }
       class div(l, r) {
22         inherits se.simpleExpressions.div(l, r)
           method eval −> Monad<Number> is override {
24             l.eval >>= { a −>
                   r.eval >>= { b −>
26                     pure(a / b) } }
           }
28     }
   }
```

Having established this framework, we then define the nodes of this monadic variant of expressions; these differ from the simple ones of § 2.1 *only* in the eval method, so the *only* content of these new classes are eval methods that override the inherited ones. The **inherits** statement on line 17 says that the methods of con in monadicExpressions are the same as those of con in simpleExpressions, *except* for the overridden method eval. The **alias** clause on the **inherits** statement provides a new name for the inherited eval, so that the overriding method body can lift the value produced by the inherited eval method into the monad. The eval method of div is similar. We ask readers concerned about type-checking this instance of "the expression problem" to suspend disbelief until Section 4.2.

Notice that the original definition of se.simpleExpressions in Section 2.1 did not specify that eval was a parameter. On the contrary: eval was defined to be a simple concrete method. Only when an inheritor of se.simpleExpressions chooses to override eval does it become a parameter. This is what we meant by *ex post facto parameterization* in the introduction.

Here is a test program parallel to the previous one: the output indicates that the results are in the monad.

```
import "monadicExpressions" as m
def e = m.monadicExpressions
def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)          // prints pure(42)
def error = e.div(e.con 1, e.con 0)
print (error.eval)           // prints pure(infinity)
```

In the remainder of this section, we show how three of Wadler's variations on this monadic interpreter — for exceptions, state, and output — can be implemented incrementally using inheritance.

2.4 Variation One: Evaluation with Exceptions

The basic idea behind adding exceptions is that the result of an evaluation is no longer always a simple value (like 42) but may also be an exception. Moreover, exceptions encountered during the evaluation of an expression terminate the evaluation and emit the exception itself as the answer. In our toy language, the only source of exceptions will be division by zero.

We have to extend our monad to deal with exceptional results. Working in Haskell, Wadler redefines the algebraic data type contained in the monad, starting from scratch. Working in Grace, we define exceptionExpressions by inheriting from monadicExpressions, and need write code only for the differences. We start by defining a new monad class raise (lines 5–8), whose bind method stops execution when an exception is encountered by immediately returning self, thus discarding any computations passed to it as its parameter k.

module "exceptionExpressions"

```
   import "monadicExpressions" as m
2
   class exceptionExpressions {
4      inherits m.monadicExpressions
       class raise(reason) −> m.Monad {
6          method asString −> String is override { "raise({reason})" }
           method >>= (k) −> m.Monad is override { self }
8      }
       class div(l, r) {
10         inherits m.monadicExpressions.div(l, r)
           method eval −> m.Monad is override {
12             l.eval >>= { a −>
                   r.eval >>= { b −>
14                     if (b == 0)
                           then { raise "Divide {a} by zero" }
16                         else { pure(a / b) } } } }
       }
18 }
```

We must also change the eval methods to raise exceptions at the appropriate time. Since the evaluation of constants cannot raise an exception, the con class is unchanged, *so we say nothing*, and use the inherited con. In contrast, evaluating a div can raise an exception, so we have to provide a new eval method for div, shown on lines 11–16. This code returns a value in the raise monad when the divisor is zero.

If you are familiar with Wadler's addition of error messages to his monadic interpreter (Wadler, 1992, §2.3) this will look quite familiar. In fact, the developments are so similar that it is easy to overlook the differences:

1. At the end of his description of the changes necessary to introduce error messages, Wadler writes: "To modify the interpreter, substitute monad E for monad M, and replace each occurrence of unitE Wrong by a suitable call to errorE. ... No other changes are required." Wadler is giving us editing instructions! In contrast, the box above represents a file of real Grace code that can be compiled and executed. It contains, if you will, not only the new definitions for the eval method and the monad, but *also the editing instructions* required to install them.
2. Not only does the boxed code represent a unit of compilation, it also represents a *unit of understanding*. We believe that it is easier to understand a complex structure like a monad with exceptions by first understanding the monad pure, and *then* understanding the exceptions. Perhaps Wadler agrees with this, for he himself uses just this form of exposition in his *paper*. But, lacking inheritance, he cannot capture this stepwise exposition in his *code*.

2.5 Variation Three: Propagating State

To illustrate the manipulation of state, Wadler keeps a count of the number of reductions; we will count the number of divisions. Each computation is given an input state in which to execute, and returns a potentially different output state; the difference between the input and output states reflect the changes to the state effected by the computation. Rather than representing $\langle result, state\rangle$ pairs with anonymous tuples, we will use Response objects with two methods, shown below.

module "response"
```
type Response = type {
    result −> Unknown
    state −> Unknown
}
class result(r) state(s) −> Response {
    method result { r }
    method state { s }
    method asString { "result({r}) state({s})" }
}
```

What changes must be made to monadicExpressions to support state? The key difference between the stateMonad and the pure monad is in the >>= method. We follow the conventional functional approach, with the contents of the monad being a function that, when applied to a state, returns a Response. The method executeIn captures this idea.

module "statefulEvaluation"

```
   import "monadicExpressions" as m
 2 import "response" as rp

 4 class statefulEvaluation {
       inherits m.monadicExpressions
 6     class stateMonad(contents:Fun) -> m.Monad {
           method asString -> String { "stateMonad({executeIn 0})" }
 8         method executeIn(s) -> rp.Response { contents.apply(s) }
           method >>= (k:Fun) -> m.Monad {
10             stateMonad { s ->
                   def resp = executeIn(s)
12                 k.apply(resp.result).executeIn(resp.state) }
           }
14     }
       method pure(a) -> m.Monad is override {
16         stateMonad { s -> rp.result(a) state(s) }
       }
18     method tally -> m.Monad {
           stateMonad { s -> rp.result(done) state(s + 1) }
20     }
       class div(l, r) is override {
22         inherits m.monadicExpressions.div(l, r)
               alias monadicEval = eval
24         method eval -> m.Monad is override {
               tally >>= { x -> monadicEval }
26         }
       }
28 }
```

The >>= method of the state monad (lines 9–13) returns a new state monad that, when executed, will first execute its contents in the given state s, then apply its argument k to the result, and then execute the contents of that monad in the threaded state. Given this definition of >>=, and a redefinition of the pure method to ensure that values are lifted into the correct monad, we do not need to redefine *any* of the tree nodes — so long as their evaluation does not involve using the state.

Of course, the *point* of this extension is to enable some evaluations to depend on state. We will follow Wadler in using state in a somewhat artificial way: the state will be a single number that counts the number of divisions. The tally method increments the count stored in the state and answers done (Grace's unit), indicating that its purpose is to cause an effect rather than to compute a result.

Naturally, counting divisions requires extending the eval operation in the div class (but not elsewhere) to do the counting. The overriding eval method on lines 24–26 first counts using tally, and then calls the inherited eval method. Notice also that the asString operation of the monad (line 7) executes the contents of the monad in the state 0, representing the initial value of the counter. This is exactly parallel to Wadler's function showS.

Here is an example program similar to the previous one:

```
import "statefulEvaluation" as se
def e = se.statefulEvaluation

def simple = e.con 34
print (simple.eval)         // prints stateMonad(result(34) state(0))

def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)         // prints stateMonad(result(42) state(2))

def error = e.div(e.con 1, e.con 0)
print (error.eval)         // prints stateMonad(result(infinity) state(1))
```

2.6 Variation Four: Output

Our final example builds up an output string, using similar techniques. We reuse monadicExpressions, this time overriding the inherited eval methods to produce output. We provide a new monad that holds both result and output, and is equipped with a bind operator that accumulates the output at each step.

module "outputEvaluation"

```
import "monadicExpressions" as m

class outputEvaluation {
    inherits m.monadicExpressions
    class con(n) {
        inherits m.monadicExpressions.con(n)
        method eval −> m.Monad {
            out "{self.asString} = {n}\n" >>= { _ −> pure(n) }
        }
    }
    class div(l, r) {
        inherits m.monadicExpressions.div(l, r)
        method eval −> m.Monad is override {
            l.eval >>= { a: Number −>
                r.eval >>= { b: Number −>
                    out "{self.asString} = {a / b}\n" >>= { _ −> pure(a / b) } } }
        }
    }
    method pure(a) −> m.Monad is override { monad(a) output "" }
    method out(s) −> m.Monad { monad(done) output(s) }
    class monad(contents') output(output') {
        method contents {contents'}
        method output {output'}
        method asString { "output monad contents: {contents} output:\n{output}" }
        method >>= (k) {
            def next = k.apply(contents)
            monad(next.contents) output(output ++ next.output)
        }
    }
}
```

Here is an example program, with its output.

```
import "outputEvaluation" as o

def e = o.outputEvaluation
def answer = e.div(e.div(e.con 1932, e.con 2), e.con 23)
print (answer.eval)
// prints output monad contents: 42 output:
// con 1932 = 1932
// con 2 = 2
// div(con 1932, con 2) = 966
// con 23 = 23
// div(div(con 1932, con 2), con 23) = 42
def error = e.div(e.con 1, e.con 0)
print (error.eval)

// prints output monad contents: infinity output:
// con 1 = 1
// con 0 = 0
// div(con 1, con 0) = infinity
```

3 The Erlang OTP Platform

The essential role that inheritance can play in *explaining* how a software system works was brought home to Black in 2011. Black was on sabbatical in Edinburgh, hosted by Phil Wadler. Black and Wadler had been discussing ways of characterizing encapsulated state — principally monads and effect systems. Erlang came up in the conversation as an example of a language in which state is handled explicitly, and Black started studying Armstrong's *Programming Erlang* (2007).

The Erlang OTP "generic server" provides properties such as transactions, scalability, and dynamic code update for arbitrary server behaviours. Transactions can be implemented particularly simply because state is explicit. Armstrong writes: "Put simply, the [generic server] solves the nonfunctional parts of the problem, while the callback solves the functional part. The nice part about this is that the nonfunctional parts of the problem (for example, how to do live code upgrades) are the same for all applications." (Armstrong, 2007, p. 286)

A reader of this section of Armstrong's book can't help but get the sense that this material is significant. In a separate italicized paragraph, Armstrong writes: "This is the most important section in the entire book, so read it once, read it twice, read it 100 times — just make sure the message sinks in."

The best way that Armstrong can find to explain this most-important-of-messages is to write a small server program in Erlang, and then generalize this program to add first transactions, and then hot code swapping. The best way that Black could find to understand this message was to re-implement Armstrong's server in another language — Smalltalk. Using Smalltalk's inheritance, he was able to reflect Armstrong's development, not as a series of *separate* programs, but in the stepwise development of a *single* program.

3.1 Armstrong's Goal

What does Armstrong seek to achieve with the generic server? In this context, a "server" is a computational engine with access to persistent memory. Servers typically run on a remote computer, and in the Erlang world the primary server is backed-up by a secondary server that takes over if the primary should fail. For uniformity with the proceeding example, we here present a simplified version of the OTP server in Grace; in addition to Armstrong's Erlang version, our code is also based on Bierman, Parkinson, and Noble's translation of the OTP server to a Java-like language (2008). For brevity, and to focus attention on the use of inheritance, our version omits name resolution, remote requests, concurrency and failover.

To be concrete, let's imagine two particular servers: a "name server" and a "calculation server". The name server remembers the locations of objects, with interface:

```
type NameServer = {
    add(name:String) place(p:Location) −> Done
    whereIs(name:String) −> Location
}
```

The name of the first method in the above type is add()place(), a two-part name with two parameters.

The calculation server acts like a one-function calculator with a memory; clear clears the memory, and add adds its argument to the memory, and stores the result in the memory as well as returning it. The calculation server has the interface

```
type CalculationServer = {
    clear −> Number
    add(e:Number) −> Number
}
```

Both of these servers maintain state; we will see later why this is relevant.

Armstrong refers to the actual server code as "the callback". His goal is to write these callbacks in a simple sequential style, but with explicit state. The generic server can then add properties such as transactions, failover and hot-swapping. The simple sequential implementation of the name server is shown on the following page.

The state of this "callback" is represented by a Dictionary object that stores the *name* → *location* mapping. The method initialState returns the initial state: a new, empty, Dictionary. The method add()place()state() is used to implement the client's add()place() method. The generic server provides the additional state argument, an object representing the callback's state. The method returns a Response (as in Section 2.5) comprising the newState dictionary, and the actual result of the operation, p. Similarly, the method whereIs()state() is used to implement the client's whereIs() method. The generic server again provides the additional state argument. This method returns a Response comprising the result of looking-up name in dict and the (unchanged) state.

module "nameServer"

```
import "response" as r

type Location = Unknown
type NameServer = {
    add(name:String) place(p:Location) −> Done
    whereIs(name:String) −> Location
}
type NsState = Dictionary<String, Location>

class callback {
    method initialState −> NsState { dictionary.empty }
    method add(name:String) place(p) state (dict:NsState) −> r.Response {
        def newState = dict.copy
        newState.at(name) put(p)
        r.result(p) state(newState)
    }
    method whereIs(name:String) state(dict:NsState) −> r.Response {
        def res = dict.at(name)
        r.result(res) state(dict)
    }
}
```

Finally, let's consider what happens if this name server callback is asked for the location of a name that is not in the dictionary. The lookup dict.at(name) will raise an exception, which the callback itself does not handle.

Notice that our *nameServer* module contains a class callback whose instances match the type

```
type Callback<S> = type {
    initialState −> S
}
```

for appropriate values of S. This is true of all server callback modules. Particular server callbacks extend this type with additional methods, all of which have a name that ends with the word state, and which take an extra argument of type S that represents their state.

3.2 The Basic Server

Our class server corresponds to Armstrong's module server1. This is the "generic server" into which is installed the "callback" that programs it to provide a particular function (like name lookup, or calculation).

A Request encapsulates the name of an operation and an argument list. The basic server implements two methods: handle(), which processes a single incoming request, and serverLoop(), which manages the request queue.

module "basicServer"

```
import "mirrors" as m

type Request = type {
    name -> String
    arguments -> List<Unknown>
}

class server(callbackName:String) {
    var callbackMirror
    var state
    startUp(callbackName)

    method startUp(name) {
        def callbackModule = m.loadDynamicModule(name)
        def callbackObject = callbackModule.callback
        callbackMirror := m.reflect(callbackObject)
        state := callbackObject.initialState
    }
    method handle(request:Request) {
        def cbMethodMirror = callbackMirror.getMethod(request.name ++ "state")
        def arguments = request.arguments ++ [state]
        def ans = cbMethodMirror.requestWithArgs(arguments)
        state := ans.state
        ans.result
    }
    method serverLoop(requestQ) {
        requestQ.do { request ->
            def res = handle(request)
            log "handle: {request.name} args: {request.arguments}"
            log "    result: {res}"
        }
    }
    method log(message) { print(message) }
}
```

A server implements three methods. Method startUp(name) loads the callback module name and initializes the server's state to that required by the newly-loaded callback.

The method handle accepts an incoming request, such as "add()place()", and appends the string "state", to obtain a method name like "add()place()state". It then requests that the callback executes its method with this name, passing it the arguments from the request and an additional state argument. Thus, a request like add "BuckinghamPalace" place "London" might be transmitted to the nameServer as add "BuckinghamPalace" place "London" state (dictionary.empty). The state component of the response would then be a dictionary containing the mapping from "BuckinghamPalace" to "London"; this new dictionary would provide the state argument for the next request.

The method serverLoop is a simplified version of Armstrong's loop/3 that omits the code necessary to receive messages and send back replies, and instead uses a local queue of messages and Grace's normal method-return mechanism.

Here is some code that exercises the basic server:

```
import "basicServer" as basic

class request(methodName)withArgs(args) {
    method name { methodName }
    method arguments { args }
}
def queue = [
    request "add()place()" withArgs ["BuckinghamPalace", "London"],
    request "add()place()" withArgs ["EiffelTower", "Paris"],
    request "whereIs()" withArgs ["EiffelTower"]
]
print "starting basicServer"
basic.server("nameServer").serverLoop(queue)
print "done"
```

To keep this illustration as simple as possible, this code constructs the requests explicitly; in a real remote server system, the requests would be constructed using reflection, or an RPC stub generator. This is why they appear as, for example, `request "add()place()" withArgs ["BuckinghamPalace", "London"]`, instead of as `add "BuckinghamPalace" place "London"`. Here is the log output:

```
starting basicServer
handle: add()place() args: [BuckinghamPalace, London]
        result: London
handle: add()place() args: [EiffelTower, Paris]
        result: Paris
handle: whereIs() args: [EiffelTower]
        result: Paris
done
```

Note that if the server callback raises an exception, it will crash the whole server.

3.3 Adding Transactions

Armstrong's server2 adds transaction semantics: if the requested operation raises an exception, the server loop continues with the *original* value of the state. In contrast, if the requested operation completes normally, the server continues with the *new* value of the state.

Lacking inheritance, the only way that Armstrong can explain this to his readers is to present the *entire text* of a new module, server2. The reader is left to compare each function in server2 with the prior version in server1. The start functions seem to be identical; the rpc functions are similar, except that the receive clause in server2 has been extended to accommodate an additional component in the reply messages. The function loop seems to be completely different.

In the Grace version, the differences are much easier to find. In the *transactionServer* module, the class server is derived from *basicServer*'s server using inheritance:

```
module "transactionServer"
import "mirrors" as mirrors
import "basicServer" as basic

type Request = basic.Request

class server(callbackName:String) {
    inherits basic.server(callbackName)
        alias basicHandle = handle

    method handle(request:Request) is override {
        try {
            basicHandle(request)
        } catch { why ->
            log "Error — server crashed with {why}"
            "!CRASH!"
        }
    }
}
```

The handle method is overridden, but *nothing else changes*. It's easy to see that the extent of the change is the addition of the try()catch() clause to the handle method.

If we now try and make bogus requests:

```
import "transactionServer" as transaction

class request(methodName)withArgs(args) {
    method name { methodName }
    method arguments { args }
}
def queue = [
    request "add()place()" withArgs ["BuckinghamPalace", "London"],
    request "add()place()" withArgs ["EiffelTower", "Paris"],
    request "whereIs()" withArgs ["EiffelTower"],
    request "boojum()" withArgs ["EiffelTower"],
    request "whereIs()" withArgs ["BuckinghamPalace"]
]
print "starting transactionServer"
transaction.server("nameServer").serverLoop(queue)
print "done"
```

they will be safely ignored:

```
starting transactionServer
handle: add()place() args: [BuckinghamPalace, London]
        result: London
handle: add()place() args: [EiffelTower, Paris]
        result: Paris
handle: whereIs() args: [EiffelTower]
        result: Paris
Error — server crashed with NoSuchMethod: no method boojum()state in mir-
ror for a callback
handle: boojum() args: [EiffelTower]
        result: !CRASH!
```

handle: whereIs() args: [BuckinghamPalace]
result: London
done

Armstrong emphasizes that the same server callback can be run under both the basic server and the transaction sever. This is also true for the Grace version, but that's not what we wish to emphasize. Our point is that inheritance makes it much easier to understand the critical differences between *basicServer* and *transactionServer* than does rewriting the whole server, as Armstrong is forced to do.

3.4 The Hot-Swap Server

Armstrong's server3 adds "hot swapping" to his server1; once again he is forced to rewrite the whole server from scratch, and the reader must compare the two versions of the code, line by line, to find the differences. Our Grace version instead adds hot swapping to the transactionServer, again using inheritance.

module "hotSwapServer"

```
import "mirrors" as mirrors
import "transactionServer" as base

type Request = base.Request

class server(callbackName:String) {
    inherits base.server(callbackName)
        alias baseHandle = handle

    method handle(request:Request) is override {
        if ( request.name == "!HOTSWAP!" ) then {
            def newCallback = request.arguments.first
            startUp(newCallback)
            "{newCallback} started."
        } else {
            baseHandle(request)
        }
    }
}
```

In *hotSwapServer*, class server overrides the handle method with a version that checks for the special request !HOTSWAP!. Other requests are delegated to the handle method inherited from *transactionServer*. Once again, it is clear that *nothing else changes*.

Now we can try to change the name server into a calculation server:

```
import "hotSwapServer" as hotSwap

class request(methodName) withArgs(args) {
    method name { methodName }
    method arguments { args }
}
def queue = [
    request "add()place()" withArgs ["EiffelTower", "Paris"],
    request "whereIs()" withArgs ["EiffelTower"],
    request "!HOTSWAP!" withArgs ["calculator"],
    request "whereIs()" withArgs ["EiffelTower"],
    request "add()" withArgs [3],
    request "add()" withArgs [4]
]
print "starting hotSwapServer"
hotSwap.server("nameServer").serverLoop(queue)
print "done"
```

Here is the output:

```
starting hotSwapServer
handle: add()place() args: [EiffelTower, Paris]
    result: Paris
handle: whereIs() args: [EiffelTower]
    result: Paris
handle: !HOTSWAP! args: [calculator]
    result: calculator started.
Error — server crashed with NoSuchMethod: no method whereIs()state in mir-
ror for a callback
handle: whereIs() args: [EiffelTower]
    result: !CRASH!
handle: add() args: [3]
    result: 3
handle: add() args: [4]
    result: 7
done
```

3.5 Summary

In summary, we can say that what Armstrong found necessary to present as a series of completely separate programs, can be conveniently expressed in Grace as one program that uses inheritance. As a consequence, the final goal — a hot-swappable server that supports transactions — cannot be found in a single, monolithic piece of code, but instead is a composition of three modules. But far from being a problem, this is an *advantage*. The presentation using inheritance lets each feature be implemented, and understood, separately. Indeed, the structure of the code mirrors quite closely the structure of Armstrong's own exposition in his book.

4 Discussion and Conclusion

We will be the first to admit that a few small examples prove nothing. Moreover, we are aware that there are places where our argument needs improvement. Here we comment on two of these.

4.1 From the Abstract to the Concrete

Inheritance can be used in many ways other than the way illustrated here. Our own experience with inheritance is that we sometimes start out with a concrete class X, and then write a new concrete class Y doing something similar. We then abstract by creating an (abstract) superclass A that contains the attributes that A and B have in common, and rewrite X and Y to inherit from A.

When programmers have a lot of experience with the domain, they might even *start out* with the abstract superclass. We did exactly this when building the Grace collections library, which contains abstract superclasses like iterable, which can be inherited by any concrete class that defines an iterator method. Syntax aside, an abstract superclass is essentially a functor returning a set of methods, and the abstract method(s) of the superclass are its *explicit* parameters. Used in this way, inheritance is little more than a convenient parameterization construct.

If we admit that this use of inheritance as a parameterization construct is common, the reader may wonder why have we used the bulk of this paper to emphasize the use of inheritance as a construct for differential programming. The answer is that we wish to capture the *essence* of inheritance. According to *The Blackwell Dictionary of Western Philosophy*, "essence is the property of a thing without which it could not be what it is." (Bunnin and Yu, 2004, p.223). In this sense, the essence of inheritance is its ability to override a concrete entity, and thus effectively turn a constant into a parameter. There is nothing at all wrong with using inheritance to share abstract entities designed for reuse, that is, to move from the abstract to the concrete. But what makes inheritance unique is its ability to create parameters out of constants — what we have called ex post facto parameterization.

4.2 What About Types?

Anyone who has read both Wadler's "The Essence of Functional Programming" and our Section 2 can hardly fail to notice that Wadler's code is rich with type information, while our code omits many type annotations. This prompts the question: can the inheritance techniques that we advocate be well-typed?

The way that inheritance is used in Section 2 — overriding the definition of the monad in descendants — poses some fundamental difficulties for typechecking. As the components of the expression class are changed to add new functionality, the types that describe those components also change. For example, the parameters l and r of the class div in module "simpleExpressions" have eval methods that answer numbers, whereas the corresponding parameters to div in module "monadicExpressions" have eval methods that answer monads. If there were other methods that used the results of eval, these would also need to be changed to maintain type correctness. Even changing a type to a subtype is not safe in general, because the type may be used to specify a requirement on a parameter as well as a property of a result.

Allowing an overriding method in a subclass to have a type different from that of the overridden method clashes with one of the sacred cows of type-theory:

"modular typechecking" (Cardelli, 1997). The idea behind modular typechecking is that each component of the program can be typechecked once and for all, regardless of how and where it is used. Since the methods of an object are generally interdependent, it seems clear that we cannot permit unconstrained changes to the type of a method after the typecheck has been performed. This runs contrary to the need to override methods to generalize functionality, as we have done in these examples.

Two paths are open to us. The first, exemplified by Ernst (2001), Bruce (2003), and Nystrom, Chong, and Myers (2004), is to devise clever restrictions that permit both useful overriding and early typechecking. The second is to question the value of modular typechecking.

While modular typechecking seems obviously desirable for a subroutine library, or for a class that we are to *instantiate*, its importance is less clear for a class that we are going to *inherit*. After all, inheritance lets us interfere with the internal connections between the methods of the inherited class, substituting arbitrary new code in place of existing code. It is quite possible to break the inherited class by improper overriding: it is up to the programmer who uses inheritance to understand the internal structure of the inherited class. In this context, running the typechecker over both the inherited code *and* the inheriting code seems like a small price to pay for the security that comes from being able to override method types and have the resulting composition be certified as type-safe.

4.3 Further Extensions

An issue we have not yet discussed is the difficulty of combining separate extensions, for example, including both output and exceptions in the expression language of Section 2. Some form of multiple inheritance would seem to be indicated, but most attempts at multiple inheritance run into difficulties when the inherited parts conflict. A particularly troublesome situation is the "diamond problem" (Bracha and Cook, 1990; Snyder, 1986), also known as "fork-join inheritance" (Sakkinen, 1989); this occurs when a class inherits a state component from the *same* base class via multiple paths.

In our view, the most promising solution seems to be to restrict what can be multiply inherited. For example, traits, as defined by Ducasse, Nierstrasz, Schärli, Wuyts, and Black (2006), consist of collections of methods that can be "mixed into" classes. Traits do not contain state; they instead access state declared elsewhere using appropriate methods. The definition of a trait can also specify "required methods" that must be provided by the client. In exchange for these modest restrictions on expressiveness, traits avoid almost all of the problems of multiple inheritance. Traits have been adopted by many recent languages; we are in the process of adding them to Grace. Delegation may provide an alternative solution, but seems to be further from the mainstream of object-oriented language development.

One of the anonymous reviewers observed that Edwards' *Subtext* (2005a) is also based on the idea that humans are better at concrete thinking than at abstraction. Subtext seeks to “decriminalize copy and paste” (Edwards, 2005b) by supporting *post hoc* abstraction. Because Subtext programs are structures in a database rather than pieces of text, the same program can be viewed in multiple ways. Thus, whether someone reading a program sees a link to an abstraction or an embedded implementation of that abstraction depends on a check-mark on a style sheet, not on a once-and-for-all decision made by the writer of the program. Moreover, the view can be changed in real time to suit the convenience of the reader. Similar ideas were explored by Black and Jones (2004).

As we have shown in this article, inheritance can lead to highly-factored code. This can be a double-edged sword. We have argued that the factoring is advantageous, because it lets the reader of a program digest it in small, easily understood chunks. But it can sometimes also be a disadvantage, because it leaves the reader with a less integrated view of the program. The ability to view a program in multiple ways can give us the best of both worlds, by allowing the chunks either to be viewed separately, or to be “flattened” into a single larger chunk without abstraction boundaries. A full exploration of these ideas requires not only that we “step away from the ASR-33” (Kamp, 2010), but that we move away from textual representations of programs altogether.

4.4 Conclusion

Introducing a new concept by means of a series of examples is a technique as old as pedagogy itself. All of us were taught that way, and teach that way. We do not start instruction in mathematics by first explaining the abstractions of rings and fields, and then introducing arithmetic on the integers as a special case. On the contrary: we introduce the abstractions only after the student is familiar with not one, but several, concrete examples.

As Baniassad and Myers (2009) have written,

> the code that constitutes a program actually forms a higher-level, program-specific language. The symbols of the language are the abstractions of the program, and the grammar of the language is the set of (generally unwritten) rules about the allowable combinations of those abstractions. As such, a program is both a language definition, and the only use of that language. This specificity means that reading a never-before encountered program involves learning a new natural language

It follows that when we write a program, we are teaching a new language. Isn’t it our duty to use all available technologies to improve our teaching? So, next time you find yourself explaining a complex program by introducing a series of simpler programs of increasing complexity, think how that *explanation*, as well as the final program, could be captured in your programming language. And if your chosen programming language does not provide the right tools, ask whether it could be improved.

References

Armstrong, J.: Programming Erlang: Software for a concurrent world. Pragmatic Bookshelf (2007)

Baniassad, E., Myers, C.: An exploration of program as language. In: OOPSLA '09 Companion. pp. 547–556 (2009)

Bierman, G.M., Parkinson, M.J., Noble, J.: UpgradeJ: incremental typechecking for class upgrades. In: ECOOP '08 — Object-Oriented Programming. pp. 235–259 (2008)

Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates (2009), http://pharobyexample.org

Black, A.P., Bruce, K.B., Homer, M., Noble, J.: Grace: the absence of (inessential) difficulty. In: Onward! '12: Proc. 12th ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software. pp. 85–98. ACM, New York, NY (2012), http://doi.acm.org/10.1145/2384592.2384601

Black, A.P., Jones, M.P.: The case for multiple views. In: Grundy, J.C., Welland, R., Stoeckle, H. (eds.) ICSE Workshop on Directions in Software Engineering Environments (WoDiSEE). (May 2004)

Bracha, G., Cook, W.: Mixin-based inheritance. In: Proc. Joint European Conf. on Object-Oriented Programming and ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications. pp. 303–311. OOPSLA/ECOOP '90, ACM Press, Ottawa, Canada (1990), http://dx.doi.org/10.1145/97946.97982

Bruce, K.B.: Some challenging typing issues in object-oriented languages. Electr. Notes Theor. Comput. Sci. 82(7), 1–29 (2003), http://dx.doi.org/10.1016/S1571-0661(04)80799-0

Bunnin, N., Yu, J.: The Blackwell Dictionary of Western Philosophy. Blackwell (2004)

Cardelli, L.: Program fragments, linking, and modularization. In: POPL '97: 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Paris, France. pp. 266–277– (Jan 1997)

Cook, W., Palsberg, J.: A denotational semantics of inheritance and its correctness. In: Conf. on Object-oriented programming systems, languages and applications. pp. 433–443. ACM Press, New Orleans, LA USA (1989)

Cook, W.R., Hill, W.L., Canning, P.S.: Inheritance is not subtyping. In: Conf. Record Seventeenth ACM Symp. on Principles of Programming Languages. pp. 125–135. San Francisco, CA, USA (1990)

Cox, B.J.: Object Oriented Programming: An Evolutionary Approach. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)

Edwards, J.: Subtext: Uncovering the simplicity of programming. In: Proc. 20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications. pp. 505–518. OOPSLA '05, ACM, New York, NY, USA (2005a), http://doi.acm.org/10.1145/1094811.1094851

Edwards, J.: Subtext: Uncovering the simplicity of programming (2005b), http://subtextual.org/demo1.html, video demonstration

Ernst, E.: Family polymorphism. In: ECOOP '2001 — Object-Oriented Programming. pp. 303–326. ECOOP '01, Springer-Verlag, London, UK, UK (2001), http://dl.acm.org/citation.cfm?id=646158.680013

Kamp, P.H.: Sir, please step away from the ASR-33! ACM Queue 8(10), 40:40–40:42 (Oct 2010), http://doi.acm.org/10.1145/1866296.1871406

Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications. pp. 99–115. OOPSLA '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1028976.1028986

Reynolds, J.C.: Definitional interpreters for higher order programming languages. In: Proc ACM 1972 Annual Conf. pp. 717–740 (1972)

Reynolds, J.C.: The essence of ALGOL. In: de Bakker, J.W., van Vliet, J.C. (eds.) Proc. International Symp. on Algorithmic Languages. pp. 345–372. North-Holland (1981), reprinted in *Algol-like Languages*, ed. P. W. O'Hearn and R.D. Tennent, vol. 1 pp. 67–88, Birkhäuser, 1997.

Sakkinen, M.: Disciplined inheritance. In: Cook, S. (ed.) ECOOP '89: Proc. Third European Conf. on Object-Oriented Programming, Nottingham, UK, July 10-14, 1989. pp. 39–56. Cambridge University Press (1989)

Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In: OOPSLA '86. pp. 38–45 (1986), also published as ACM SIGPLAN Notices , vol. 21, November 1986

Wadler, P.: The essence of functional programming. In: Conf. Record Nineteenth ACM Symp. on Principles of Programming Languages. pp. 1–14. ACM Press, Albuquerque, NM (1992)

Wadler, P.: The first monad tutorial (December 2013), http://homepages.inf.ed.ac.uk/wadler/papers/yow/monads-scala.pdf

A Implementing Queues

Here is a further example, suggested by Ross Tate: a queue object. It's simpler than the others, and is probably the best one to start the exposition.

The initial version of the queue uses a linear array `elements` to store the queue elements. When the queue fills up, we simply allocate a larger array and copy the elements across.

module "queue"

```
// implements a queue using an array to store the elements

class empty {
    // answers a new empty queue.  The contents are in
    // elements[firstIx], elements[firstIx+1],  ... elements[endIx − 1]

    def initialSize = 4
    var elements := primitiveArray.new(initialSize)
    var firstIx := 0
    var endIx := 0

    method size { endIx − firstIx }
    method isEmpty { endIx == firstIx }
    method capacity is confidential { elements.size }
    method add(e) {
        if (isFull) then { makeMoreRoom }
        elements.at (endIx) put (e)
        endIx := increment (endIx)
        self
    }
    method remove {
        if (size == 0) then { NoSuchObject.raise "can't remove from an empty queue" }
        def result = elements.at(firstIx)
        firstIx := increment(firstIx)
        result
    }
    method asString {
        var s := "⊢"
        usedIndicesDo { ix −>
            s := "{s} {elements.at(ix)} ⇽"
        }
        s
    }
    method asDebugString {
        "q[{firstIx}..{endIx−1}]#{capacity} {size}:{asString}"
    }
    method makeMoreRoom is confidential {
        def newElements = primitiveArray.new(capacity ∗ 2)
        usedIndicesDo { i −>
            newElements.at(i) put (elements.at(i))
        }
        elements := newElements
    }
    method isFull is confidential { endIx == capacity }
    method usedIndicesDo (action) is confidential {
        var i := firstIx
        repeat (size) times {
            action.apply (i)
            i := increment (i)
        }
    }
    method increment(ix) is confidential { ix + 1 }
}
```

This clearly wastes space at the start of the array — space that can never be reused. An obvious optimization is to copy the elements into the new array starting at the bottom (index 0), rather than copying them straight across.

module "queue+slide"

```
// implements a queue using an array to store the elements

import "queue" as originalQueue

class empty {
    // Similar to originalQueue except that, when my contents are copied into a larger elements
    // array, we slide them to the bottom, rather than coping them into their former locations.

    inherit originalQueue.empty

    method makeMoreRoom is confidential, override {
        def newElements = primitiveArray.new(capacity * 2)
        var j := 0
        usedIndicesDo { i ->
            newElements.at(j) put (elements.at(i))
            j := increment(j)
        }
        elements := newElements
        firstIx := 0
        endIx := j
    }
}
```

Once we have seen the idea of sliding the elements down to the bottom of the array, we realize that we can also apply it to recycle the empty locations when the queue contents reaches the top of the array, even without allocating a larger one.

module "queue+recycle"

```
// implements a queue using an array to store the elements

import "queue" as originalQueue

class empty {
    // Similar to originalQueue except that, before allocating a larger elements array, we see
    // if it is worthwhile to recycle the now−unused space at the bottom of the current array.

    inherit originalQueue.empty
        alias enlarge = makeMoreRoom

    method makeMoreRoom is confidential, override {
        def threshold = 2
        if ((capacity − size) > threshold)
            then { slideInPlace } else { enlarge }
    }

    method slideInPlace is confidential {
        usedIndicesDo { i −>
            elements.at(i − firstIx) put (elements.at(i))
        }
        endIx := endIx − firstIx
        firstIx := 0
    }
}
```

This works fine, but prompts us to wonder why we are doing all the copying. We can get the same effect, with no copying, by treating elements as a circular array. This brings us to the final version:

module "queue+wrap"

```
// implements a queue using an array to store the elements

import "queue" as originalQueue

class empty {
    // answers a new empty queue.  The contents are in elements[firstIx], elements[firstIx+1], ...,
    // elements[endIx − 1], but there is no assumption that endIx <= startIx.  Instead, elements
    // is treated as a circular array, and indexing is modulo its capacity.  When "full",
    // endIx == startIx − 1 (mod capacity); this enables us to distinguish this case from "empty",
    // when endIx == startIx (mod capacity).

    inherit originalQueue.empty

    method size is override { (endIx − firstIx) % capacity }
    method increment(ix) is override, confidential { (ix + 1) % capacity }
    method isFull is override, confidential { endIx == ((firstIx − 1) % capacity) }
}
```

Here is a simple test suite:

```
dialect "minitest"
// test four different implementations of a queue.  They all support the same add
// and remove operations, but differ in the way that they allocate and reuse space.
// These differences are revealed by requesting asDebugString after the test sequence.
import "queue" as qOrig
import "queue+slide" as qSlide
import "queue+recycle" as qRecycle
import "queue+wrap" as qWrap

[qOrig, qSlide, qRecycle, qWrap].do { queue ->

    testSuite {
        def q = queue.empty
        test "empty" by {
            assert (q.size) shouldBe 0
            assert (q.asString) shouldBe "⊢"
        }
        test "add 3" by {
            q.add "first"
            q.add "second".add "third"
            assert (q.size) shouldBe 3
            assert (q.asString) shouldBe "⊢ first ⇽ second ⇽ third ⇽"
        }
        test "add and remove" by {
            q.add "first"
            q.add "second".add "third"
            assert (q.remove) shouldBe "first"
            assert (q.remove) shouldBe "second"
            assert (q.remove) shouldBe "third"
            assert (q.size) shouldBe 0
            assert {q.remove} shouldRaise (NoSuchObject)
        }
        test "+20 −18" by {
            using (q) add 20 remove 18 add 0 remove 0
            assert (q.asString) shouldBe "⊢ 19 ⇽ 20 ⇽"
        }
        test "+4, −3, +5, −5" by {
            using (q) add 4 remove 3 add 5 remove 5
            assert (q.asString) shouldBe "⊢ 9 ⇽"
        }
        test "+8, −6, +4, −5" by {
            using (q) add 8 remove 6 add 4 remove 5
            assert (q.asString) shouldBe "⊢ 12 ⇽"
        }
        test "+7, −5, +4, −5" by {
            using (q) add 7 remove 5 add 4 remove 5
            assert (q.asString) shouldBe "⊢ 11 ⇽"
        }
    }
}
```

```
method using (q) add (a1) remove (r1) add (a2) remove (r2) is confidential {
    // a helper method that for the test cases above.

    (1..a1).do { i −> q.add (i) }
    assert (q.size) shouldBe (a1)
    (1..r1).do { i −> assert (q.remove) shouldBe (i) }
    assert (q.size) shouldBe (a1−r1)
    ((a1+1)..(a1+a2)).do { i −> q.add (i) }
    assert (q.size) shouldBe (a1 + a2 − r1)
    ((r1+1)..(r1+r2)).do { i −> assert (q.remove) shouldBe (i) }
    assert (q.size) shouldBe (a1 + a2 − r1 −r2)
    print "after +{a1}, −{r1}, +{a2}, −{r2}: q = {q.asDebugString}"
}
```

The output from these tests is as follows:

```
after +20, −18, +0, −0: q = q[18..19]#32 2:⊢ 19 ⇠ 20 ⇠
after +4, −3, +5, −5: q = q[8..8]#16 1:⊢ 9 ⇠
after +8, −6, +4, −5: q = q[11..11]#16 1:⊢ 12 ⇠
after +7, −5, +4, −5: q = q[10..10]#16 1:⊢ 11 ⇠
7 run, 0 failed, 0 errors
after +20, −18, +0, −0: q = q[18..19]#32 2:⊢ 19 ⇠ 20 ⇠
after +4, −3, +5, −5: q = q[5..5]#8 1:⊢ 9 ⇠
after +8, −6, +4, −5: q = q[5..5]#16 1:⊢ 12 ⇠
after +7, −5, +4, −5: q = q[5..5]#16 1:⊢ 11 ⇠
7 run, 0 failed, 0 errors
after +20, −18, +0, −0: q = q[18..19]#32 2:⊢ 19 ⇠ 20 ⇠
after +4, −3, +5, −5: q = q[5..5]#8 1:⊢ 9 ⇠
after +8, −6, +4, −5: q = q[5..5]#8 1:⊢ 12 ⇠
after +7, −5, +4, −5: q = q[5..5]#8 1:⊢ 11 ⇠
7 run, 0 failed, 0 errors
after +20, −18, +0, −0: q = q[18..19]#32 2:⊢ 19 ⇠ 20 ⇠
after +4, −3, +5, −5: q = q[0..0]#8 1:⊢ 9 ⇠
after +8, −6, +4, −5: q = q[11..11]#16 1:⊢ 12 ⇠
after +7, −5, +4, −5: q = q[2..2]#8 1:⊢ 11 ⇠
7 run, 0 failed, 0 errors
```

The output from asDebugString shows the interval of elements that is in use, the capacity of elements, the size of the queue, and the queue contents. So, for example, *q[18..19]#32 2:⊢ 19 ⇠ 20 ⇠* means that elements 18 and 19 are occupied, that the capacity of elements is 32, and that the queue contains 2 values, 19 at the head and 20 at the back. Notice that each refinement reduces memory usage, except for the penultimate test, where *queue+wrap* uses an array of capacity 16 for a test in which *queue+recycle* has managed with an array of capacity 8. In this test case *queue+recycle* completely fills elements, which queue +wrap cannot do, because it preserves at least one unused element to enable it to distinguish between full and empty queues.