The Essence of Inheritance

Andrew P. Black

Joint work with Kim Bruce & James Noble

This talk

- Inheritance as an aid to *human understanding* of programs
  - *Not* about the formal properties of inheritance.
  - *Not* about types

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.

Inheritance = Parametrization of Generators

(OOPSLA 1989)

A Denotational Semantics of Inheritance and its Correctness

William Cook*
Department of Computer Science
Box 1910 Brown University
Providence, RI 02912, USA
wrc@cs.brown.edu

Jens Palsberg
Computer Science Department
Aarhus University
Ny Munkegade, DK-8000
Aarhus C, Denmark
palsberg@daimi.dk

Abstract

This paper presents a denotational model of inheritance. The model is based on an intuitive motivation of the purpose of inheritance. The correctness of the model is demonstrated by proving it equivalent to an operational semantics of inheritance based upon the method-lookup algorithm of object-oriented languages. Although it was originally developed to explain inheritance in object-oriented languages, the model shows that inheritance is a general mechanism that may be applied to any form of recursive definition.

1 Introduction

Inheritance is one of the central concepts in object-oriented programming. Despite its importance, there seems to be a lack of consensus on the proper way to describe inheritance. This is evident from the following review of various formalizations of inheritance that have been proposed.

The concept of *prefixing* in Simula [5], which evolved into the modern concept of inheritance, was defined in terms of textual concatenation of program blocks. However, this definition was informal, and only partially accounted for more sophisticated aspects of prefixing like the pseudo-variable this and virtual operations.

The most precise and widely used definition of inheritance is given by the operational semantics of object-oriented languages. The canonical operational semantics is the "method lookup" algorithm of Smalltalk:

> When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found. ...
>
> When a method contains a message whose receiver is self, the search for the method for that message begins in the instance's class, regardless of which class contains the method containing self. ...
>
> When a message is sent to super, the search for a method ... begins in the superclass of the class containing the method. The use of super allows a method to access methods defined in a superclass even if the methods have been overridden in the subclasses. [6, pp. 61–64]

Unfortunately, such operational definitions do not necessarily foster intuitive understanding. As a result, insight into the proper use and purpose of inheritance is often gained only through an "Aha!" experience [1].

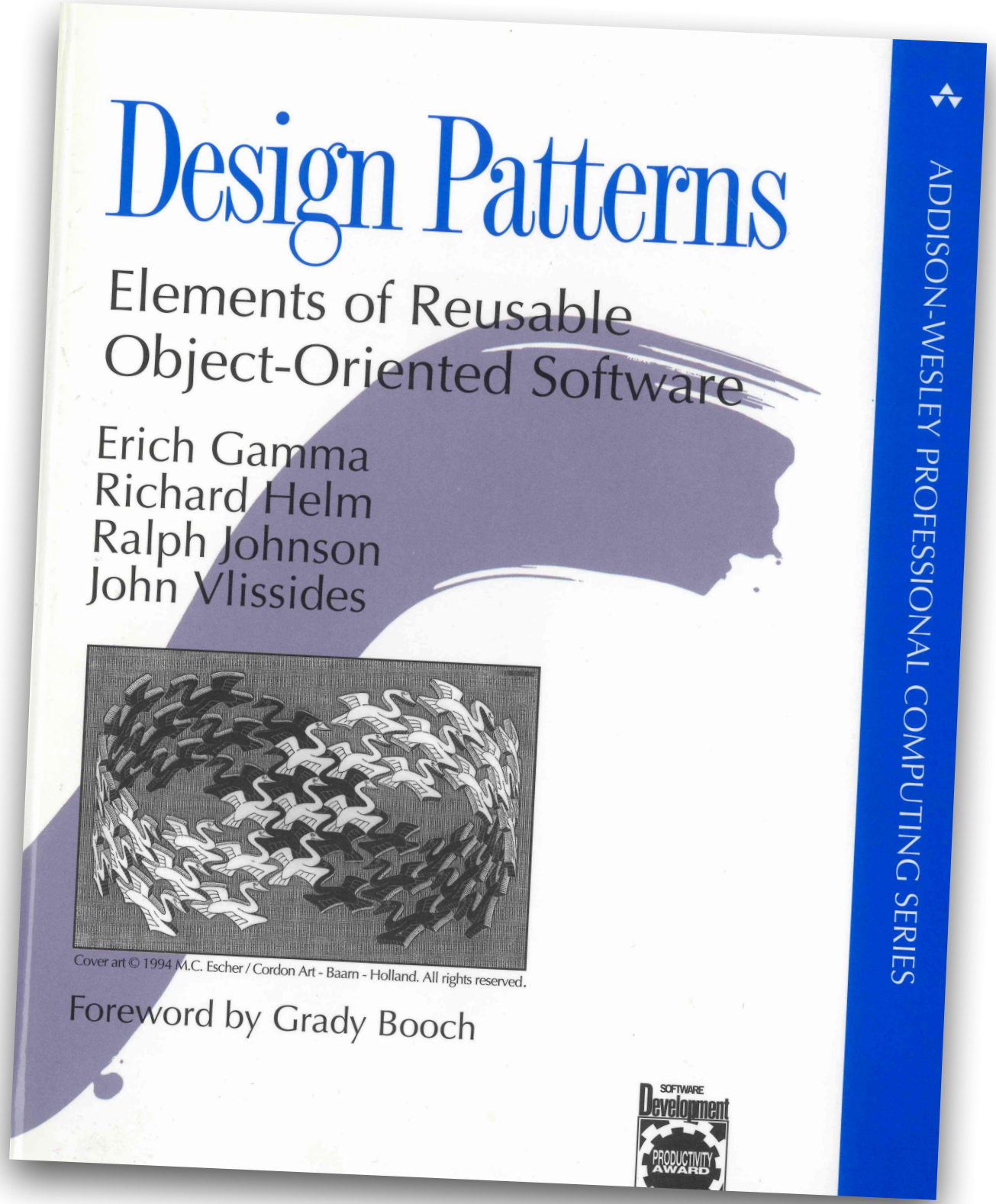
Cardelli [2] identifies inheritance with the subtype relation on record types: "a record type τ is a subtype (written $\leq$) of a record type τ' if τ has all the fields of τ', and possibly more, and the common fields of τ and τ' are in the $\leq$ relation." His work shows that a sound type-checking algorithm exists for strongly-typed, statically-scoped languages with inheritance, but it doesn't give their dynamic semantics. More recently, McAllister and Zabih [9] suggested a system of "boolean classes" similar to inheritance as used in knowledge representation. Stein [16] focused on shared attributes and methods. Minsky and Rozenshtein [10] characterized inheritance by "laws" regulating message sending. Although they express various aspects of inheritance, none of these presentations are convincing because they provide no verifiable evidence that the formal model corresponds to the form of inheritance actually used in object-oriented

* *Current address*: Hewlett-Packard Laboratories, P.O. Box 10490 Palo Alto, CA 94303-0969, cook@hplabs.hp.com.

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.

The Object-Oriented gang do not always help

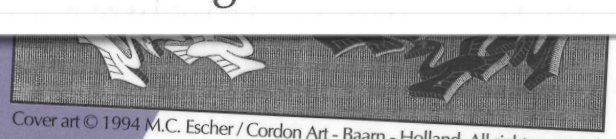

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
SOFTWARE Development PRODUCTIVITY AWARD
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns

That leads us to our second principle of object-oriented design:

Favor object composition over class inheritance.

Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition. But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.



Foreword by Grady Booch

SOFTWARE Development PRODUCTIVITY AWARD

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Problem:

- Explaining the value of inheritance
  - especially to functional programmers

higher order

Abstract: good Concrete: *not so good*

Abstract

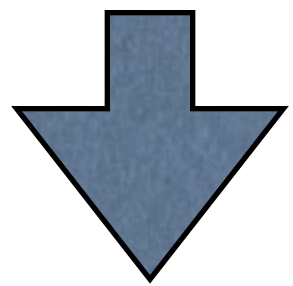
Wait!

Can you please give me an example?


Abstrac
oncrete

Inheritance is good at doing this!

Concrete



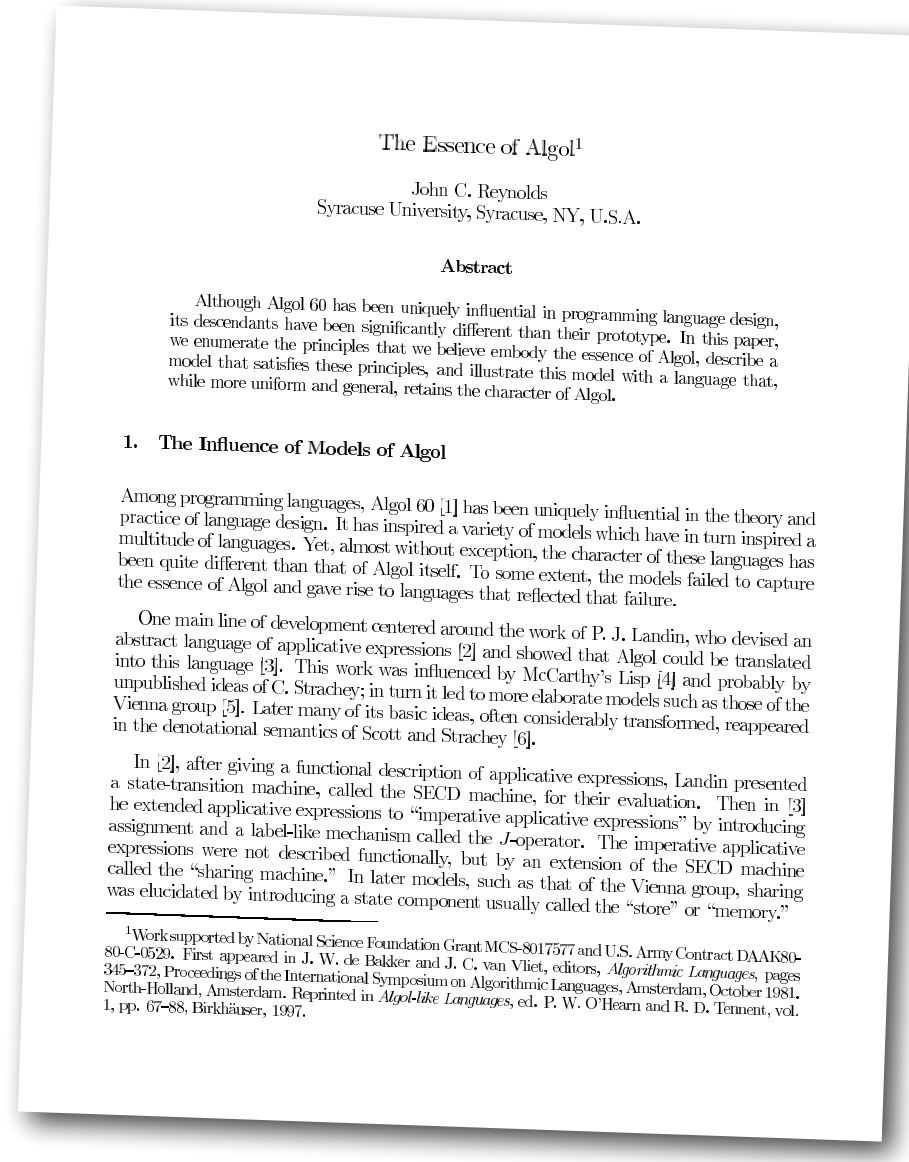
Abstract

In the beginning…

Reynolds: The Essence of Algol (1981)

Proceedings of the International Symposium on Algorithmic Languages

values ≠ meanings



The Essence of Algol[1]

John C. Reynolds
Syracuse University, Syracuse, NY, U.S.A.

Abstract

Although Algol 60 has been uniquely influential in programming language design, its descendants have been significantly different than their prototype. In this paper, we enumerate the principles that we believe embody the essence of Algol, describe a model that satisfies these principles, and illustrate this model with a language that, while more uniform and general, retains the character of Algol.

1. The Influence of Models of Algol

Among programming languages, Algol 60 [1] has been uniquely influential in the theory and practice of language design. It has inspired a variety of models which have in turn inspired a multitude of languages. Yet, almost without exception, the character of these languages has been quite different than that of Algol itself. To some extent, the models failed to capture the essence of Algol and gave rise to languages that reflected that failure.

One main line of development centered around the work of P. J. Landin, who devised an abstract language of applicative expressions [2] and showed that Algol could be translated into this language [3]. This work was influenced by McCarthy's Lisp [4] and probably by unpublished ideas of C. Strachey; in turn it led to more elaborate models such as those of the Vienna group [5]. Later many of its basic ideas, often considerably transformed, reappeared in the denotational semantics of Scott and Strachey [6].

In [2], after giving a functional description of applicative expressions, Landin presented a state-transition machine, called the SECD machine, for their evaluation. Then in [3] he extended applicative expressions to "imperative applicative expressions" by introducing assignment and a label-like mechanism called the *J*-operator. The imperative applicative expressions were not described functionally, but by an extension of the SECD machine called the "sharing machine." In later models, such as that of the Vienna group, sharing was elucidated by introducing a state component usually called the "store" or "memory."

[1]Work supported by National Science Foundation Grant MCS-8017577 and U.S. Army Contract DAAK80-80-C-0529. First appeared in J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Proceedings of the International Symposium on Algorithmic Languages, Amsterdam, October 1981. North-Holland, Amsterdam. Reprinted in *Algol-like Languages*, ed. P. W. O'Hearn and R. D. Tennent, vol. 1, pp. 67–88, Birkhäuser, 1997.

2. There are two fundamentally different kinds of type: *data types*, each of which denotes a set of values appropriate for certain variables and expressions, and *phrase types*, each of which denotes a set of meanings appropriate for certain identifiers and phrases.

This syntactic distinction reflects that fact that in Algol values (which can be assigned to variables) are inherently different from meanings (which can be denoted by identifiers and phrases, and passed as parameters). Thus Algol-like languages contradict the principle of completeness [9].

Moreover, in Algol itself data types are limited to unstructured types such as **integer** or **Boolean**, while structuring mechanisms such as procedures and arrays are only applicable to phrase types.

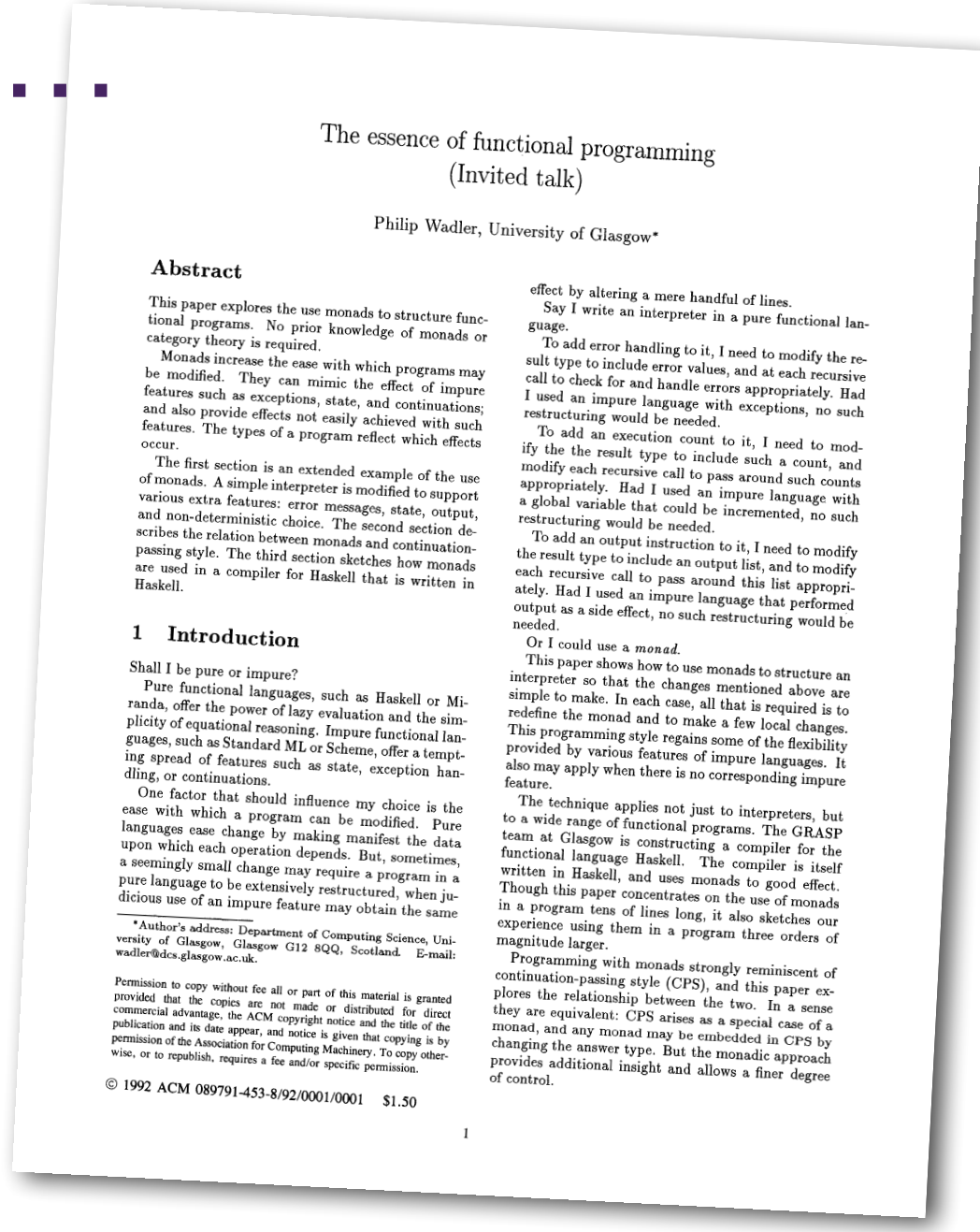
Reynolds: The Essence of Algol

This motivated …

Wadler: The Essence of Functional Programming

PoPL 1992

values ≠ computations



The essence of functional programming
(Invited talk)

Philip Wadler, University of Glasgow*

Abstract

This paper explores the use monads to structure functional programs. No prior knowledge of monads or category theory is required.

Monads increase the ease with which programs may be modified. They can mimic the effect of impure features such as exceptions, state, and continuations; and also provide effects not easily achieved with such features. The types of a program reflect which effects occur.

The first section is an extended example of the use of monads. A simple interpreter is modified to support various extra features: error messages, state, output, and non-deterministic choice. The second section describes the relation between monads and continuation-passing style. The third section sketches how monads are used in a compiler for Haskell that is written in Haskell.

1 Introduction

Shall I be pure or impure?

Pure functional languages, such as Haskell or Miranda, offer the power of lazy evaluation and the simplicity of equational reasoning. Impure functional languages, such as Standard ML or Scheme, offer a tempting spread of features such as state, exception handling, or continuations.

One factor that should influence my choice is the ease with which a program can be modified. Pure languages ease change by making manifest the data upon which each operation depends. But, sometimes, a seemingly small change may require a program in a pure language to be extensively restructured, when judicious use of an impure feature may obtain the same effect by altering a mere handful of lines.

Say I write an interpreter in a pure functional language.

To add error handling to it, I need to modify the result type to include error values, and at each recursive call to check for and handle errors appropriately. Had I used an impure language with exceptions, no such restructuring would be needed.

To add an execution count to it, I need to modify the the result type to include such a count, and modify each recursive call to pass around such counts appropriately. Had I used an impure language with a global variable that could be incremented, no such restructuring would be needed.

To add an output instruction to it, I need to modify the result type to include an output list, and to modify each recursive call to pass around this list appropriately. Had I used an impure language that performed output as a side effect, no such restructuring would be needed.

Or I could use a *monad*.

This paper shows how to use monads to structure an interpreter so that the changes mentioned above are simple to make. In each case, all that is required is to redefine the monad and to make a few local changes. This programming style regains some of the flexibility provided by various features of impure languages. It also may apply when there is no corresponding impure feature.

The technique applies not just to interpreters, but to a wide range of functional programs. The GRASP team at Glasgow is constructing a compiler for the functional language Haskell. The compiler is itself written in Haskell, and uses monads to good effect. Though this paper concentrates on the use of monads in a program tens of lines long, it also sketches our experience using them in a program three orders of magnitude larger.

Programming with monads strongly reminiscent of continuation-passing style (CPS), and this paper explores the relationship between the two. In a sense they are equivalent: CPS arises as a special case of a monad, and any monad may be embedded in CPS by changing the answer type. But the monadic approach provides additional insight and allows a finer degree of control.

*Author's address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: wadler@dcs.glasgow.ac.uk.

The essence of functional programming (Invited talk)

Philip Wadler, University of Glasgow*

Abstract

This paper explores the use monads to structure functional programs. No prior knowledge of monads or category theory is required.

Monads increase the ease with which programs may be modified. They can mimic the effect of impure features such as exceptions, state, and continuations; and also provide effects not easily achieved with such features. The types of a program reflect which effects occur.

The first section is an extended example of the use of monads. A simple interpreter is modified to support various extra features: error messages, state, output, and non-deterministic choice. The second section describes the relation between monads and continuation-passing style. The third section sketches how monads are used in a compiler for Haskell that is written in Haskell.

1 Introduction

Shall I be pure or impure?

Pure functional languages, such as Haskell or Miranda, offer the power of lazy evaluation and the simplicity of equational reasoning. Impure functional languages, such as Standard ML or Scheme, offer a tempt-

effect by altering a mere handful of lines.

Say I write an interpreter in a pure functional language.

To add error handling to it, I need to modify the result type to include error values, and at each recursive call to check for and handle errors appropriately. Had I used an impure language with exceptions, no such restructuring would be needed.

To add an execution count to it, I need to modify the the result type to include such a count, and modify each recursive call to pass around such counts appropriately. Had I used an impure language with a global variable that could be incremented, no such restructuring would be needed.

To add an output instruction to it, I need to modify the result type to include an output list, and to modify each recursive call to pass around this list appropriately. Had I used an impure language that performed output as a side effect, no such restructuring would be needed.

Or I could use a *monad*.

This paper shows how to use monads to structure an interpreter so that the changes mentioned above are simple to make. In each case, all that is required is to redefine the monad and to make a few local changes. This programming style regains some of the flexibility provided by various features of

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 then later on abstracting over a concrete element. This abstraction step is *not* performed by editing the concrete program to introduce a new parameter. That is what 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.

Black, Bruce, and Noble: Wadlerfest 2016

inheritance = *ex post facto* parameterization

[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 then later on abstracting over a concrete element. This abstraction

Shall I be Abstract or Concrete?

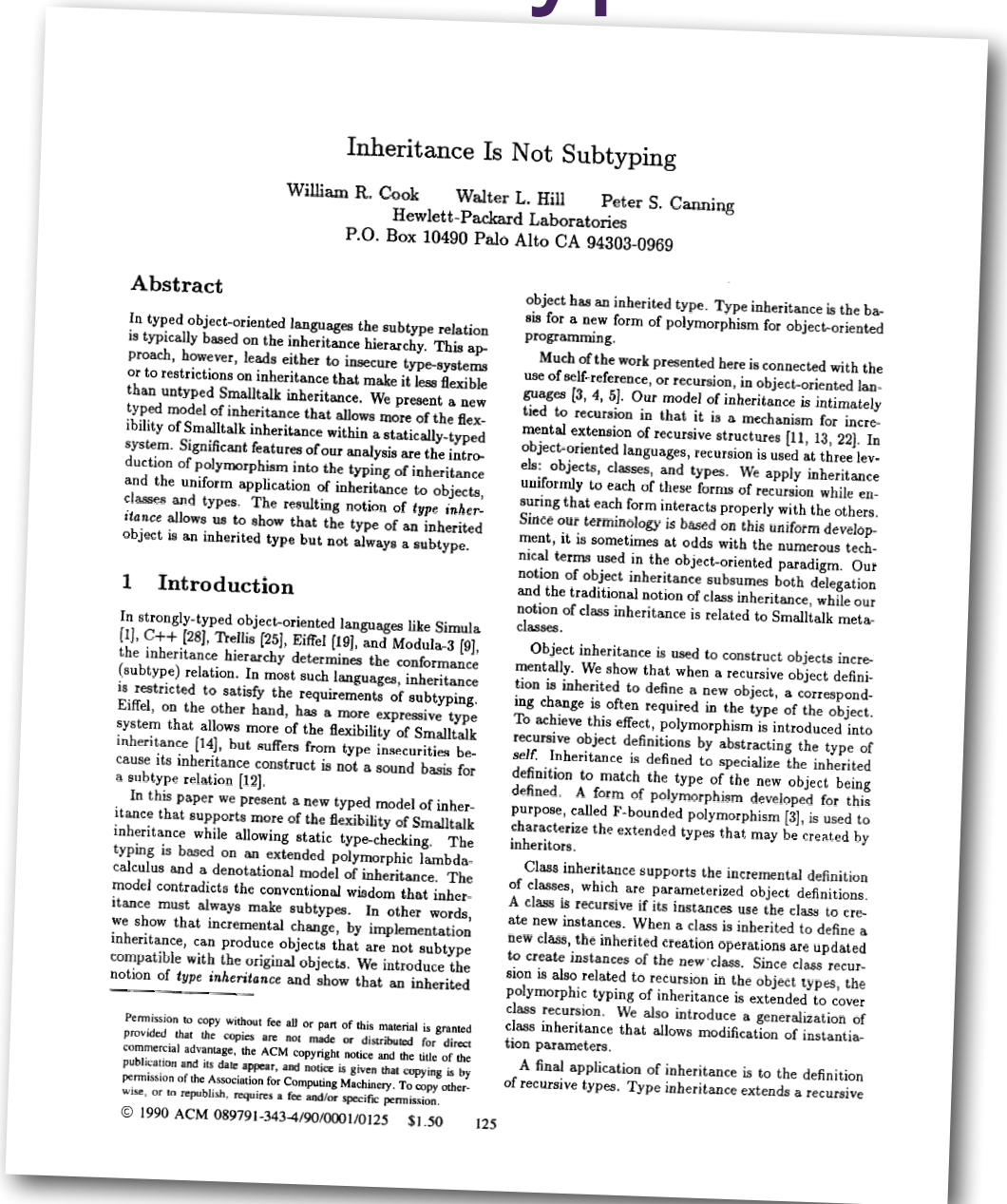
- Abstract programs are more general, more potential for reuse
- Concrete programs are simpler, solve the problem at hand more directly
- Inheritance lets us avoid this unsatisfying choice

Inheritance isn't about types

Inheritance ≠ Subtyping

Thank you, Cook & colleagues (1990)

- I'm not going to talk about types
- Examples will be in *Grace*



Inheritance Is Not Subtyping

William R. Cook Walter L. Hill Peter S. Canning
Hewlett-Packard Laboratories
P.O. Box 10490 Palo Alto CA 94303-0969

Abstract

In typed object-oriented languages the subtype relation is typically based on the inheritance hierarchy. This approach, however, leads either to insecure type-systems or to restrictions on inheritance that make it less flexible than untyped Smalltalk inheritance. We present a new typed model of inheritance that allows more of the flexibility of Smalltalk inheritance within a statically-typed system. Significant features of our analysis are the introduction of polymorphism into the typing of inheritance and the uniform application of inheritance to objects, classes and types. The resulting notion of *type inheritance* allows us to show that the type of an inherited object is an inherited type but not always a subtype.

1 Introduction

In strongly-typed object-oriented languages like Simula [1], C++ [28], Trellis [25], Eiffel [19], and Modula-3 [9], the inheritance hierarchy determines the conformance (subtype) relation. In most such languages, inheritance is restricted to satisfy the requirements of subtyping. Eiffel, on the other hand, has a more expressive type system that allows more of the flexibility of Smalltalk inheritance [14], but suffers from type insecurities because its inheritance construct is not a sound basis for a subtype relation [12].

In this paper we present a new typed model of inheritance that supports more of the flexibility of Smalltalk inheritance while allowing static type-checking. The typing is based on an extended polymorphic lambda-calculus and a denotational model of inheritance. The model contradicts the conventional wisdom that inheritance must always make subtypes. In other words, we show that incremental change, by implementation inheritance, can produce objects that are not subtype compatible with the original objects. We introduce the notion of *type inheritance* and show that an inherited object has an inherited type. Type inheritance is the basis for a new form of polymorphism for object-oriented programming.

Much of the work presented here is connected with the use of self-reference, or recursion, in object-oriented languages [3, 4, 5]. Our model of inheritance is intimately tied to recursion in that it is a mechanism for incremental extension of recursive structures [11, 13, 22]. In object-oriented languages, recursion is used at three levels: objects, classes, and types. We apply inheritance uniformly to each of these forms of recursion while ensuring that each form interacts properly with the others. Since our terminology is based on this uniform development, it is sometimes at odds with the numerous technical terms used in the object-oriented paradigm. Our notion of object inheritance subsumes both delegation and the traditional notion of class inheritance, while our notion of class inheritance is related to Smalltalk metaclasses.

Object inheritance is used to construct objects incrementally. We show that when a recursive object definition is inherited to define a new object, a corresponding change is often required in the type of the object. To achieve this effect, polymorphism is introduced into recursive object definitions by abstracting the type of *self*. Inheritance is defined to specialize the inherited definition to match the type of the new object being defined. A form of polymorphism developed for this purpose, called F-bounded polymorphism [3], is used to characterize the extended types that may be created by inheritors.

Class inheritance supports the incremental definition of classes, which are parameterized object definitions. A class is recursive if its instances use the class to create new instances. When a class is inherited to define a new class, the inherited creation operations are updated to create instances of the new class. Since class recursion is also related to recursion in the object types, the polymorphic typing of inheritance is extended to cover class recursion. We also introduce a generalization of class inheritance that allows modification of instantiation parameters.

A final application of inheritance is to the definition of recursive types. Type inheritance extends a recursive

Inheritance isn't about "accidental" reuse

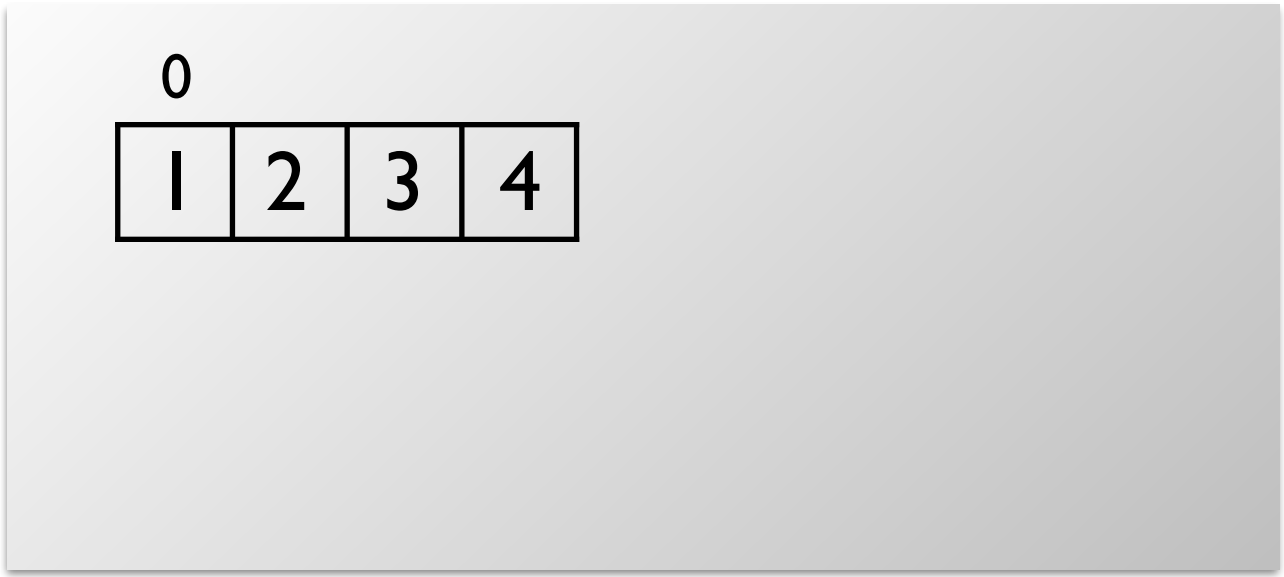
- Highly unlikely that object that not designed for reuse can be reused
  - by inheritance
  - or by any other mechanism!
- Can be refactored to facilitate reuse

Three Examples

- In the paper:
  - Evaluating Expressions (Interpreter)
    - with and without various monads
  - The Erlang OTP Platform
- In this talk:
  - Mutable Queues

A Simple Mutable Queue

```
var numberQ := queue.empty
```



A Simple Queue

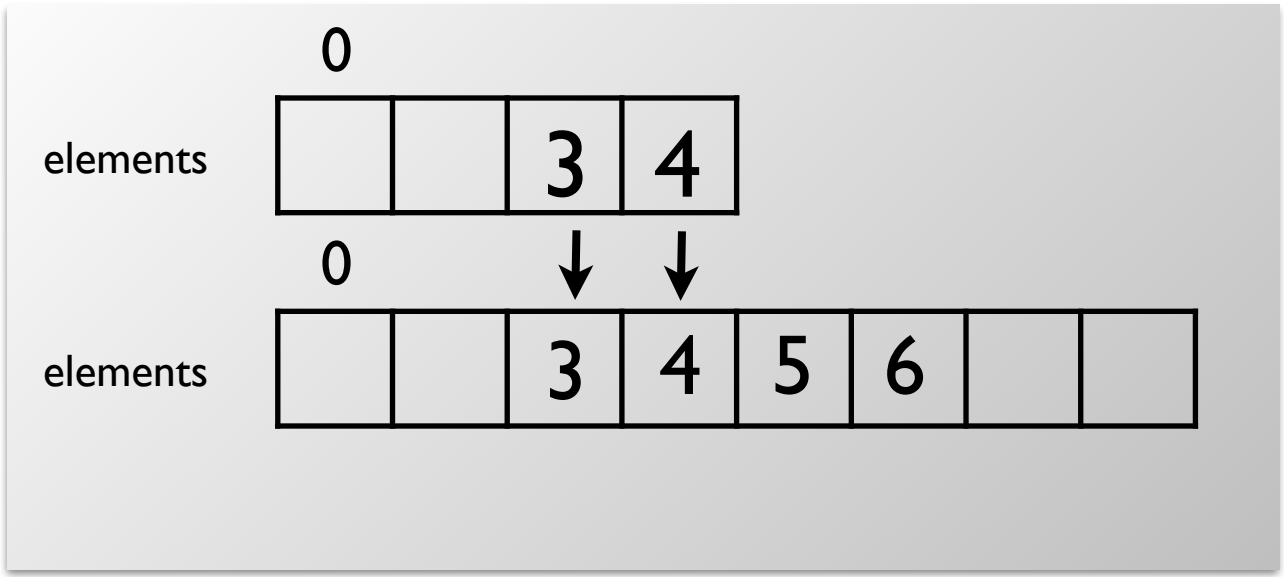
module "queue"

```
// 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 {
```

```
22          if (size == 0) then { NoSuchObject.raise "can't remove from an empty queue" }
            def result = elements.at(firstIx)
24          firstIx := increment(firstIx)
            result
26      }
        method asString {
28          var s := "⊢"
            usedIndicesDo { ix ->
30              s := "{s} {elements.at(ix)} ⇠"
            }
32          s
        }
34      method asDebugString {
            "q[{firstIx}..{endIx−1}]#{capacity} {size}:{asString}"
36      }
        method makeMoreRoom is confidential {
38          def newElements = primitiveArray.new(capacity * 2)
            usedIndicesDo { i ->
40              newElements.at(i) put (elements.at(i))
            }
42          elements := newElements
        }
44      method isFull is confidential { endIx == capacity }
        method usedIndicesDo (action) is confidential {
46          var i := firstIx
            repeat (size) times {
48              action.apply (i)
                i := increment (i)
50          }
        }
52      method increment(ix) is confidential { ix + 1 }
}
```

making room when full

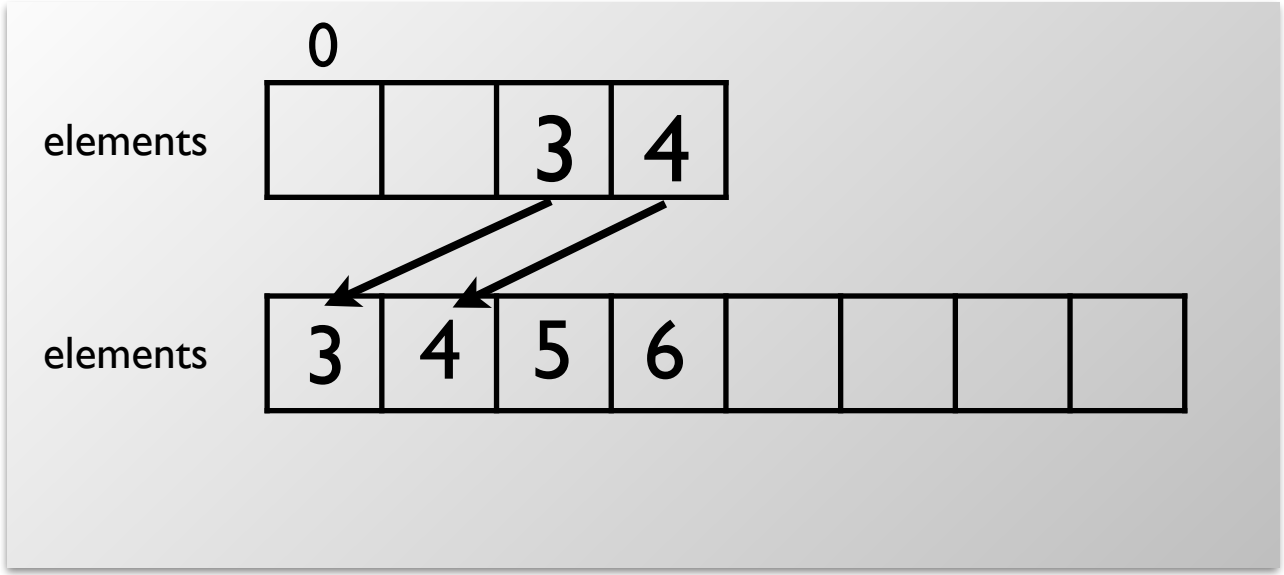
```
method makeMoreRoom is confidential {
    def newElements = primitiveArray.new(capacity * 2)
    usedIndicesDo { i ->
        newElements.at(i) put (elements.at(i))
    }
    elements := newElements
}
```



But ...

- This implementation wastes space at the start of the internal array
- An obvious optimization is to “slide down” the element when copying into the new array

A Better Plan




```
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
}
```

How to install the better plan?

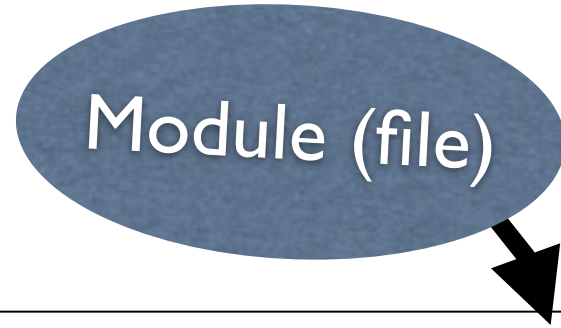
- How do we combine these code fragments?

```
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
}
```

How to install the better plan?

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
    }
}
```



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
    }
}
```

What to notice:

- Inheritance combines new code with “editing instructions” that say where to put it.
- The part being replaced was not originally declared to be a parameter
  - inheritance is ex post facto parameterization
- Inheritance lets us focus on the changes

Contrast with Wadler

2.3 Variation one: Error messages

To add error messages to the interpreter, define the following monad.

```
data  E a           =  Suc a | Err String

unitE a             =  Suc a
errorE s            =  Err s

(Suc a) `bindE` k   =  k a
(Err s) `bindE` k   =  Err s

showE (Suc a)   =  "Success: " ++ showval a
showE (Err s)   =  "Error: " ++ s
```

Each function in the interpreter either returns normally by yielding a value of the form `Suc a`, or indicates an error by yielding a value of the form `Err s` where `s` is an error message. If `m :: E a` and `k :: a -> E b` then `m `bindE` k` acts as strict postfix application: if `m` succeeds then `k` is applied to the successful result; if `m` fails then so does the application. The show function displays either the successful result or the error message.

To modify the interpreter, substitute monad `E` for monad `M`, and replace each occurrence of `unitE Wrong` by a suitable call to `errorE`. The only occurrences are in `lookup`, `add`, and `apply`.

- Inheritance provides a *packaging* mechanism for deltas
  - Inheritance = code + editing instructions

Both super- and subclass are units of *understanding*

- How do you explain a complex artifact?
  - You don't: you start with a simple one, and gradually add the complexities, one at a time
- This is what Wadler does in *Essence of Functional Programming*
- This is what Armstrong does in *Programming Erlang*
- This is what I do when I teach a class

… and it's probably what you do too.

Consequence

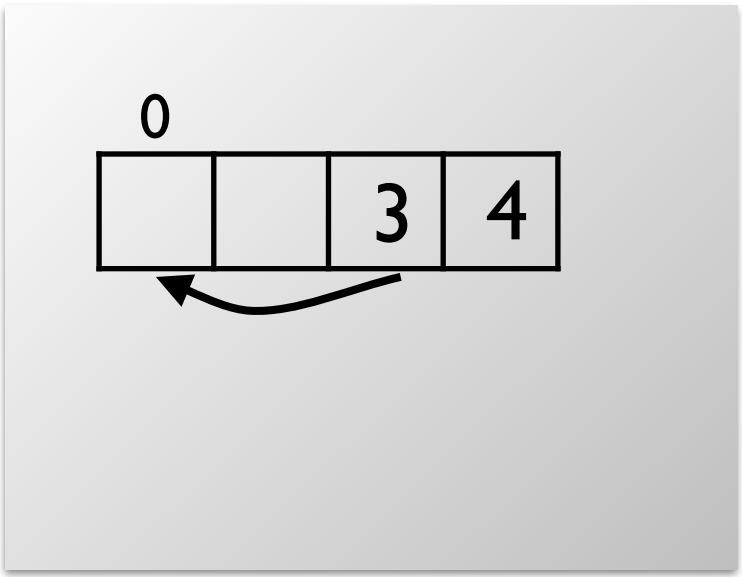
- You can’t see the whole object in one place
- True!
  - the behaviour of an object defined using inheritance is distributed through the inheritance hierarchy
- This is a feature, not a problem

Meanwhile, somewhere in Britain …

Back to the queue

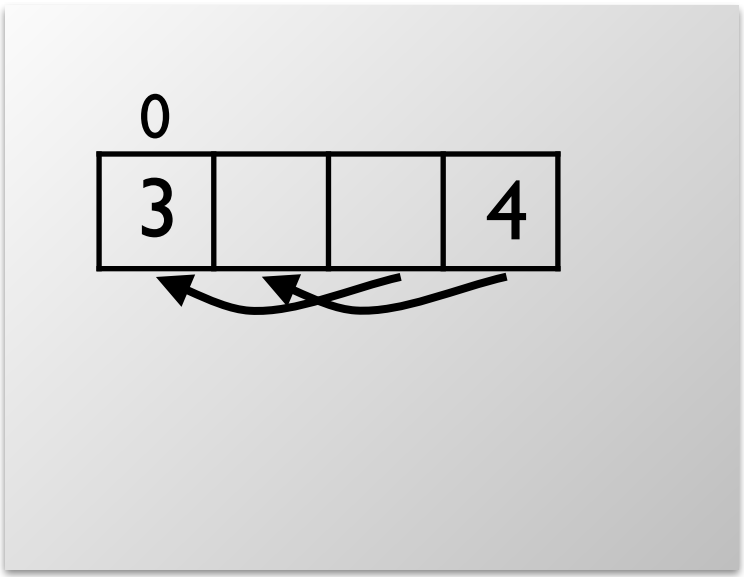
Recycling Space

- Once we see the idea of *sliding* elements to the bottom,
- We should ask: why allocate a larger array at all?



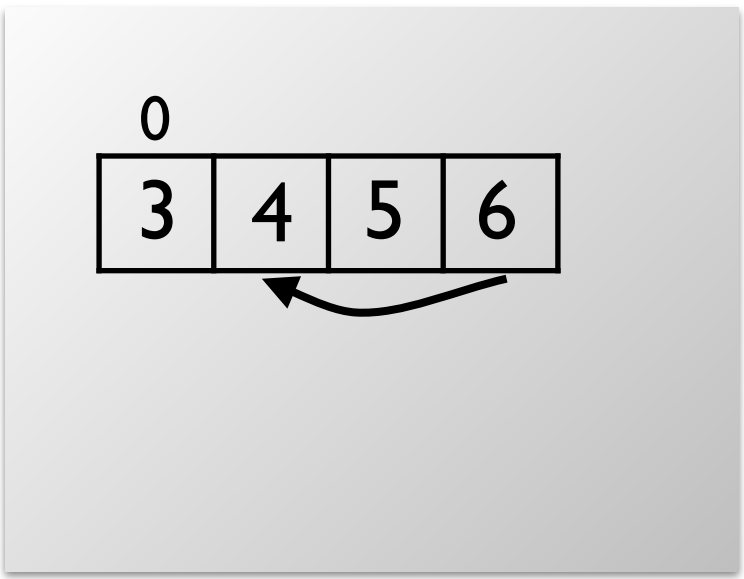
Recycling Space

- Once we see the idea of *sliding* elements to the bottom,
- We should ask: why allocate a larger array at all?



Recycling Space

- Once we see the idea of *sliding* elements to the bottom,
- We should ask: why allocate a larger array at all?



- Can we add this feature to the original queue using inheritance?

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
    }
}
```

Why Slide?

- The price of recycling space is seems to be sliding.
- But it's not: we can treat elements as a circular array

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) }
}
```

- Three method overrides implement the change

```
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"
```

```
[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 output:

qOrig

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

Test output:

qOrig

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

qSlide

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

Test output:

qOrig

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

qSlide

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

qRecycle

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

Test output:

qOrig

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

qSlide

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

qRecycle

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

qWrap

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

What about “Accidental Reuse”?

- I’m *not* claiming that inheritance supports “accidental reuse”
- Usually, code must be refactored to provide the hooks for an inheriting object to override.

My Changes

```
    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
    }
```

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

    method size { endIx - firstIx }
    method capacity is confidential
    { elements.size }
    method add(e) {
        if (endIx == elements.size) then
     { makeMoreRoom }
        elements.at(endIx) put (e)
        endIx := endIx + 1
        self
    }
    method remove {
        if (size == 0) then { NoSuchObject.raise
     "can't remove from an empty queue" }
        def result = elements.at(firstIx)
        firstIx := firstIx + 1
        result
    }
    method asString {
        var s := ""
        (firstIx..(endIx-1)).reversed.do { i ->
            s := "{s} → {elements.at(i)}"
        }
        s ++ " ]"
```

Adds several helper methods

```
    method makeMoreRoom is confidential {
        def newElements =
     primitiveArray.new(capacity * 2)
        usedIndicesDo { i ->
            newElements.at(i) put (elements.at(i))
        }
        elements := newElements
    }
```

```
    method makeMoreRoom is confidential {
        def newElements =
     primitiveArray.new(capacity * 2)
        (firstIx..(endIx-1)).do { 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 }
}
```

These changes introduce *intention-revealing method names*.
They improve communication as well as enabling inheritance

Armstrong's Explanation of Open Telecom Platform

OTP = framework for building scalable, fault-tolerant distributed systems

The Pragmatic Programmers

Chapter 16

OTP Introduction

OTP stands for the Open Telecom Platform. The name is actually misleading, because OTP is far more general than you might think. It's an application operating system and a set of libraries and procedures used for building large-scale, fault-tolerant, distributed applications. It was developed at the Swedish telecom company Ericsson and is used within Ericsson for building fault-tolerant systems.[1]

Joe Armstrong

Key idea: separate concerns

- OTP provides *behaviors* such as a "generic server"
  - generic server supports fault tolerance, transactions, hot-swapping of code, …
- Application programmer provides specific functionality in a *callback*
  - callback is simple, sequential code

Example Callbacks

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

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

Implemented with *Explicit* 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)
    }
}
```

Implemented with *Explicit* 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)
    }
}
```

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})" }
}
```

Here is the plan of this chapter:

1. Write a small client-server program in Erlang.
2. Slowly generalize this program and add a number of features.
3. Move to the real code.

16.1 The Road to the Generic Server

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.

We're going to write four little servers called server1, server2..., each slightly different from the last. The goal is to totally separate the non-functional parts of the problem from the functional parts of the problem. That last sentence probably didn't mean much to you now, but don't worry—it soon will. Take a deep breath....

Joe Armstrong

Basic Server

Server 1: The Basic Server

Here's our first attempt. It's a little server that we can parameterize with a callback module:

Adding Transactions

Server 2: A Server with Transactions

Here's a server that crashes the client if the query in the server results in an exception:

server2.erl

```
-module(server2).
-export([start/2, rpc/2]).
```

This one gives you "transaction semantics" in the server—it loops with the *original value* of State if an exception was raised in the handler function. But if the handler function succeeded, then it loops with the value of NewState provided by the handler function.

…and then Armstrong re-writes the *whole server*

Using inheritance, we need specify only the differences:

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!"
        }
    }
}
```

and so it goes on …

Server 3: A Server with Hot Code Swapping

Now we'll add hot code swapping:

server3.erl

```
-module(server3).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name,
             spawn(fun() -> loop(Name,Mod,Mod:init()) end)).
```

…and then Armstrong re-writes the *whole server* once again

Using inheritance, we override just one method:

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)
        }
    }
}
```

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) }
}
```

module "basicServer"

```
import "mirrors" as m

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

class
    va
    va
    st

    m

    m

    m

    m
    log
    log
    }
}
method log(message) { print(message) }
}
```

module "transactionServer"

```
import "mirrors" as mirrors
impo

type

class
    in

    m
    m

    }
    m
}
```

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)
        }
    }
}
```

Client code

```
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"
```

Client code

```
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"
```

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

Client Code

```
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"
```

```
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
ror for a callback
handle: whereIs() args: [EiffelTower]
        result: !CRASH!
handle: add() args: [3]
        result: 3
handle: add() args: [4]
        result: 7
done
```

Summary

- Armstrong wrote a series of separate server modules, duplicating code
  - Readers must diff to understand the changes
- Inheritance lets us write one basic server
  - Each derived server becomes a module that inherits from the basic server
  - Changes are manifest as method overrides
- Each feature can be implemented, and

Our Thesis:

- The Essence of Inheritance is that it lets us go from the concrete to the abstract
- It does this using *ex post facto* parameterization: taking a constant and turning it into a parameter

Essence

"Essence is the property of a thing without which it could not be what it is."

Blackwell Dictionary of Western Philosophy

- Our claim: the *essence* of inheritance is its ability to override a concrete entity, and thus effectively turn a constant into a parameter
- No other construct in programmingdom does that

Why “Essence” ?

- Inheritance is often used in other ways,
  - e.g., to go from the abstract to the concrete
- But used in this way, we are *explicit* about what the parameters are
  - method-placeholders labelled **abstract** or **required**
  - no more than a clumsy parametrization mechanism [Cook & Palsberg 1989]

Conclusion

The code that constitutes a program actually forms a higher-level, program-specific language. … 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

Baniassad and Myers [2009]
An exploration of program as language