On Understanding Data Abstraction ... Revisited

William R. Cook
The University
of Texas at Austin

Dedicated to P. Wegner

2

Objects

????

Abstract Data Types

3

Warnings!

4

No "Objects Model the Real World"

5

No Inheritance

6

No Mutable State

7

No Subtyping!

8

Interfaces as types

9

Not Essential

(very nice but not essential)

10


discuss
inheritance
later

Abstraction

12

13

Visible

Hidden

14

Procedural Abstraction

```
bool f(int x) { ... }
```


15

Procedural Abstraction

int → bool

16

(one kind of)

Type Abstraction

$\forall T.\mathrm{Set}[T]$

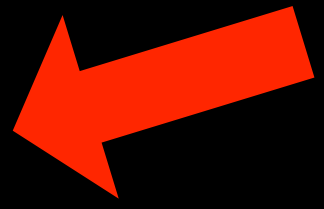
17

Abstract Data Type

signature Set
  empty : Set
  insert : Set, Int → Set
  isEmpty : Set → Bool
  contains : Set, Int → Bool

18

Abstract Data Type



Abstract

signature Set

empty	: Set
insert	: Set, Int → Set
isEmpty	: Set → Bool
contains	: Set, Int → Bool

19

Type
+
Operations

20

ADT Implementation

abstype Set = List of Int

```
  empty              = []
  insert(s, n)       = (n : s)
  isEmpty(s)         = (s == [])
  contains(s, n)     = (n ∈ s)
```


21

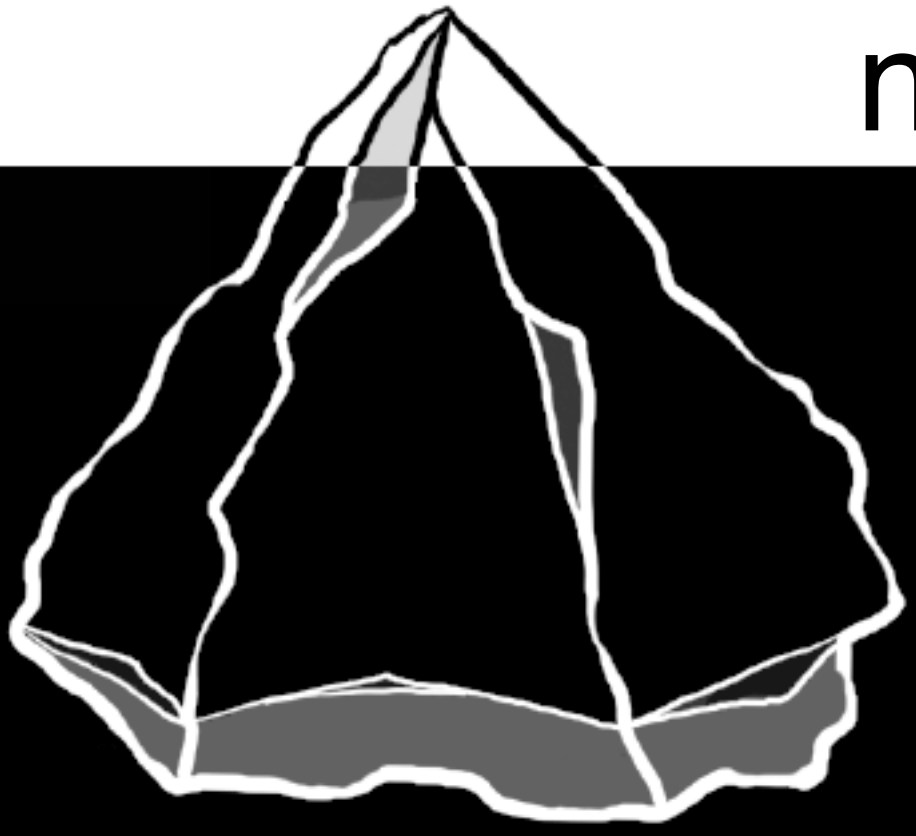
Using ADT values

```
def x:Set = empty
def y:Set = insert(x, 3)
def z:Set = insert(y, 5)
print( contains(z, 2) )==> false
```


22

23

Visible name: Set



Hidden representation: List of Int

24

$$
\begin{aligned}
&\text{ISetModule} = \exists \text{Set}.\{ \\
&\quad \text{empty} \quad : \text{Set} \\
&\quad \text{insert} \quad : \text{Set}, \text{Int} \to \text{Set} \\
&\\
&\quad \text{isEmpty} \quad : \text{Set} \to \text{Bool} \\
&\quad \text{contains} \quad : \text{Set}, \text{Int} \to \text{Bool} \\
&\}
\end{aligned}
$$

25

Natural!

26

just like
built-in types

27

Mathematical
Abstract Algebra

28

Type Theory

$\exists x.P$

(existential types)

29

Abstract Data Type

=

Data Abstraction

30

Right?

31

$$S = \{ 1, 3, 5, 7, 9 \}$$

32

Another way

33

$$P(n) = \text{even}(n) \;\&\; 1 \leq n \leq 9$$

34

$$S = \{ 1, 3, 5, 7, 9 \}$$

$$P(n) = \text{even}(n) \;\&\; 1 \leq n \leq 9$$

35

Sets as characteristic functions

36

```
type Set =
    Int → Bool
```


37

Empty =

λn. false

38

Insert(s, m) =

$\lambda n.\ (n{=}m) \vee s(n)$

39

Using them is easy

```
def x:Set = Empty
def y:Set = Insert(x, 3)
def z:Set = Insert(y, 5)
print( z(2) ) ==> false
```


40

So What?

41

Flexibility

42

set of all
even numbers

43

Set ADT:
Not Allowed!

44

or…
break open ADT & change representation

45

set of
even numbers
as a
function?

46

Even =

$\lambda n.\ (n \bmod 2 = 0)$

47

Even interoperates

```
def x:Set = Even
def y:Set = Insert(x, 3)
def x:Set = Insert(y, 5)
print( z(2) )
```

==> true

48

Sets-as-functions are objects!

49

No type abstraction required!

type Set = Int → Bool

50

multiple
methods?

sure...

51

```
interface Set {
  contains: Int → Bool
  isEmpty: Bool
}
```


52

What about Empty and Insert?

(they are classes)

53

```
class Empty {
  contains(n) { return false;}
  isEmpty()   { return true;}
}
```


54

```
class Insert(s, m) {
  contains(n) { return (n=m)
                    ∨ s.contains(n) }
  isEmpty() { return false }
}
```


55

Using Classes

```
def x:Set = Empty()
def y:Set = Insert(x, 3)
def z:Set = Insert(y, 5)
print( z.contains(2) ) ==> false
```


56

An object
is
the set of observations
that
can be made upon it

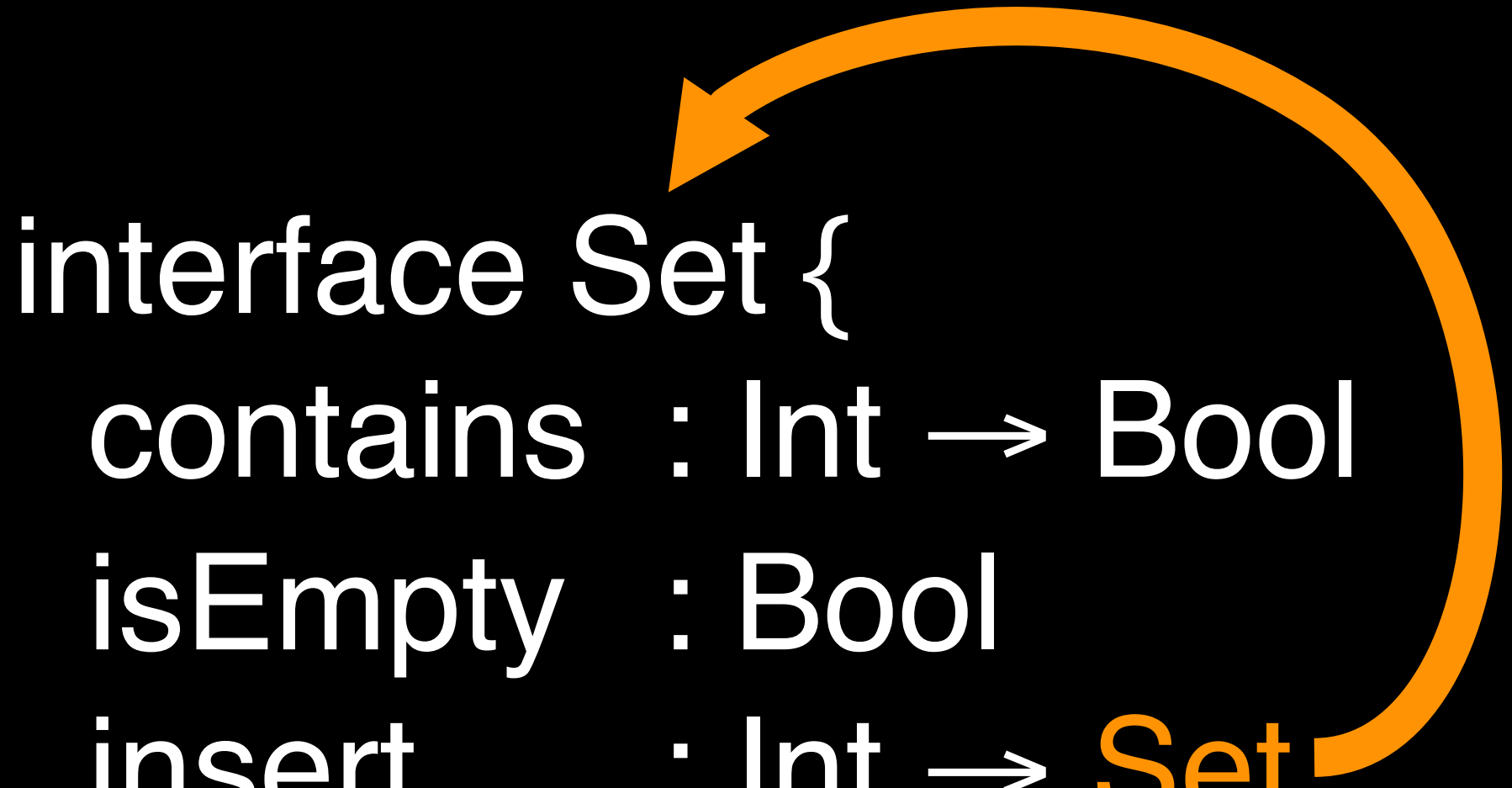
57

Including more methods

58

```
interface Set {
  contains  : Int → Bool
  isEmpty   : Bool
  insert    : Int → Set
}
```

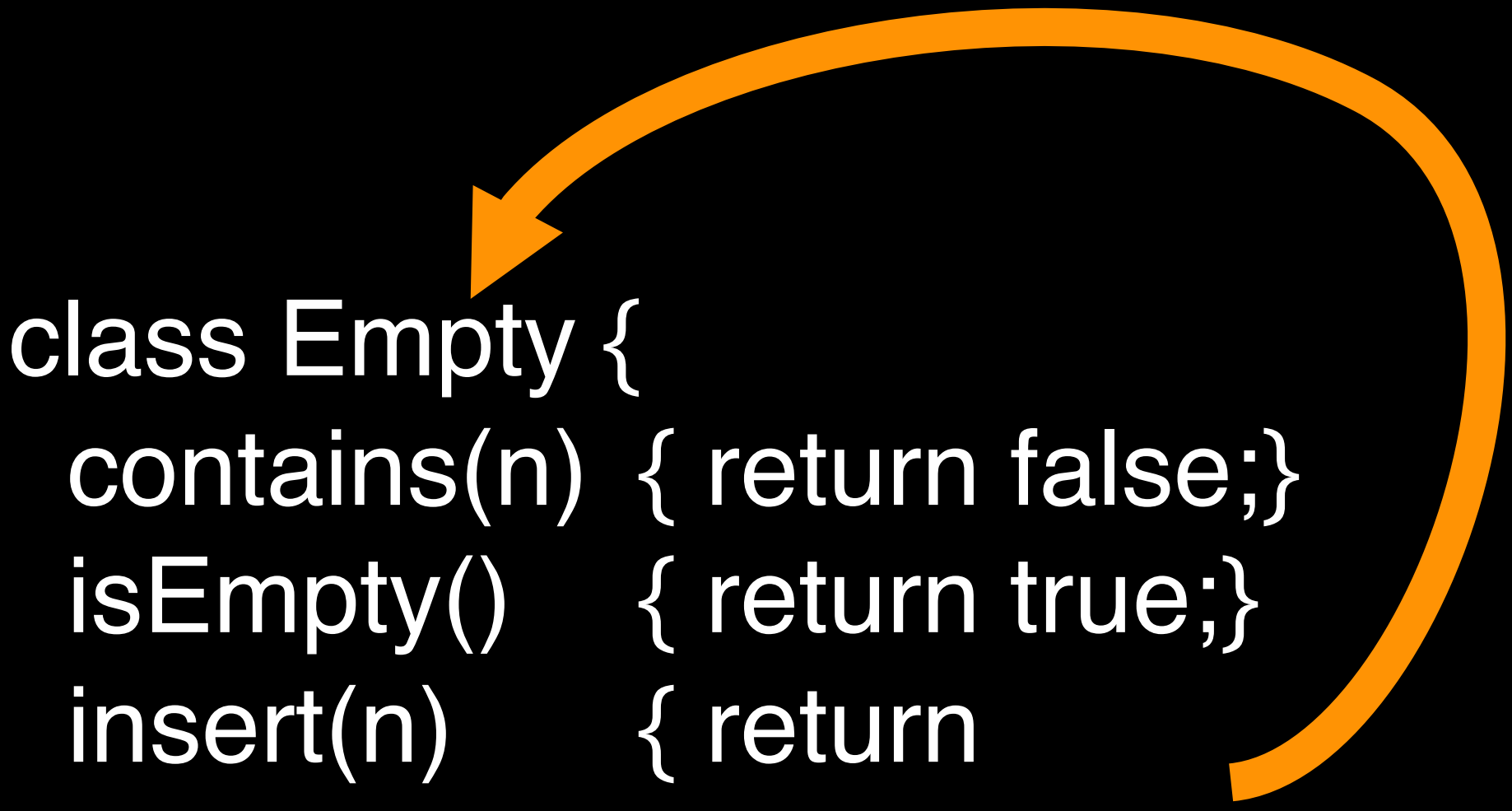

59


interface Set {
contains : Int → Bool
isEmpty : Bool
insert : Int → Set
}
Type
Recursion

60

```
class Empty {
  contains(n)  { return false;}
  isEmpty()    { return true;}
  insert(n)    { return
                   Insert(this, n);}
}
```


61



Value Recursion

62

Using objects

```
def x:Set = Empty
def y:Set = x.insert(3)
def z:Set = y.insert(5)
print( z.contains(2) )==> false
```


63

Autognosis

64

Autognosis

(Self-knowledge)

65

Autognosis

An object can access other objects only through public interfaces

66

operations
on
multiple objects?

67

union of two sets

68

```
class Union(a, b) {
  contains(n) { a.contains(n)
                ∨ b.contains(n); }
  isEmpty()   { a.isEmpty(n)
                ∧ b.isEmpty(n); }
  ...
}
```


69

```
interface Set {
  contains: Int → Bool
  isEmpty: Bool
  insert      : Int → Set
  union       : Set → Set
}
```

Complex Operation

(binary)

70

intersection
of
two sets
??

71

```
class Intersection(a, b) {
  contains(n) { a.contains(n)
                ∧ b.contains(n); }

  isEmpty()   { ? no way! ? }
  ...
}
```


72

Autognosis: Prevents some operations (complex ops)

73

Autognosis: Prevents some optimizations (complex ops)

74

Inspecting two representations & optimizing operations on them are easy with ADTs

75

Objects are fundamentally different from ADTs

76

Object Interface
(recursive types)

Set = {
  isEmpty : Bool
  contains : Int → Bool
  insert   : Int → Set
  union   : Set → Set
}
Empty : Set
Insert : Set x Int → Set
Union : Set x Set → Set

ADT
(existential types)

SetImpl = ∃ Set . {
  empty : Set
  isEmpty : Set → Bool
  contains : Set, Int → Bool
  insert : Set, Int → Set
  union : Set, Set → Set
}

77

Operations/Observations

	s	
	Empty	Insert(s', m)
isEmpty(s)	true	false
contains(s, n)	false	n=m ∨ contains(s', n)
insert(s, n)	false	Insert(s, n)
union(s, s'')	isEmpty(s'')	Union(s, s'')

78

ADT Organization

	s	
	Empty	Insert(s', m)
isEmpty(s)	true	false
contains(s, n)	false	n=m ∨ contains(s', n)
insert(s, n)	false	Insert(s, n)
union(s, s'')	isEmpty(s'')	Union(s, s'')

79

OO Organization

	s	
	Empty	Insert(s', m)
isEmpty(s)	true	false
contains(s, n)	false	n=m ∨ contains(s', n)
insert(s, n)	false	Insert(s, n)
union(s, s'')	isEmpty(s'')	Union(s, s'')

80

Objects are fundamental (too)

81

Mathematical functional representation of data

82

Type Theory

$\mu X.P$

(recursive types)

83

ADTs require
a
static type system

84

Objects work well with or without static typing

85

“Binary” Operations?

Stack, Socket, Window, Service, DOM, Enterprise Data, …

86

Objects are very higher-order

(functions passed as data and returned as results)

87

Verification

88

ADTs: construction

Objects: observation

89

ADTs: induction

Objects: coinduction complicated by: callbacks, state

90

Objects are designed to be as difficult as possible to verify

91

Simulation

One object can simulate another! (identity is bad)

92

Java

93

What is a type?

94

Declare variables

Classify values

95

Class as type

=> representation

96

Class as type

=> ADT

97

Interfaces as type

=> behavior
pure objects

98

Harmful!

instanceof *Class*
(*Class*) exp
Class x;

99

Object-Oriented subset of Java: class name used only after "new"

100

It's not an accident that "int" is an ADT in Java

101

Smalltalk

102

```
class True
  ifTrue: a ifFalse: b
      ^a

class False
  ifTrue: a ifFalse: b
      ^b
```


103

True =

λ a . λ b .

a

False =

λ a . λ b .

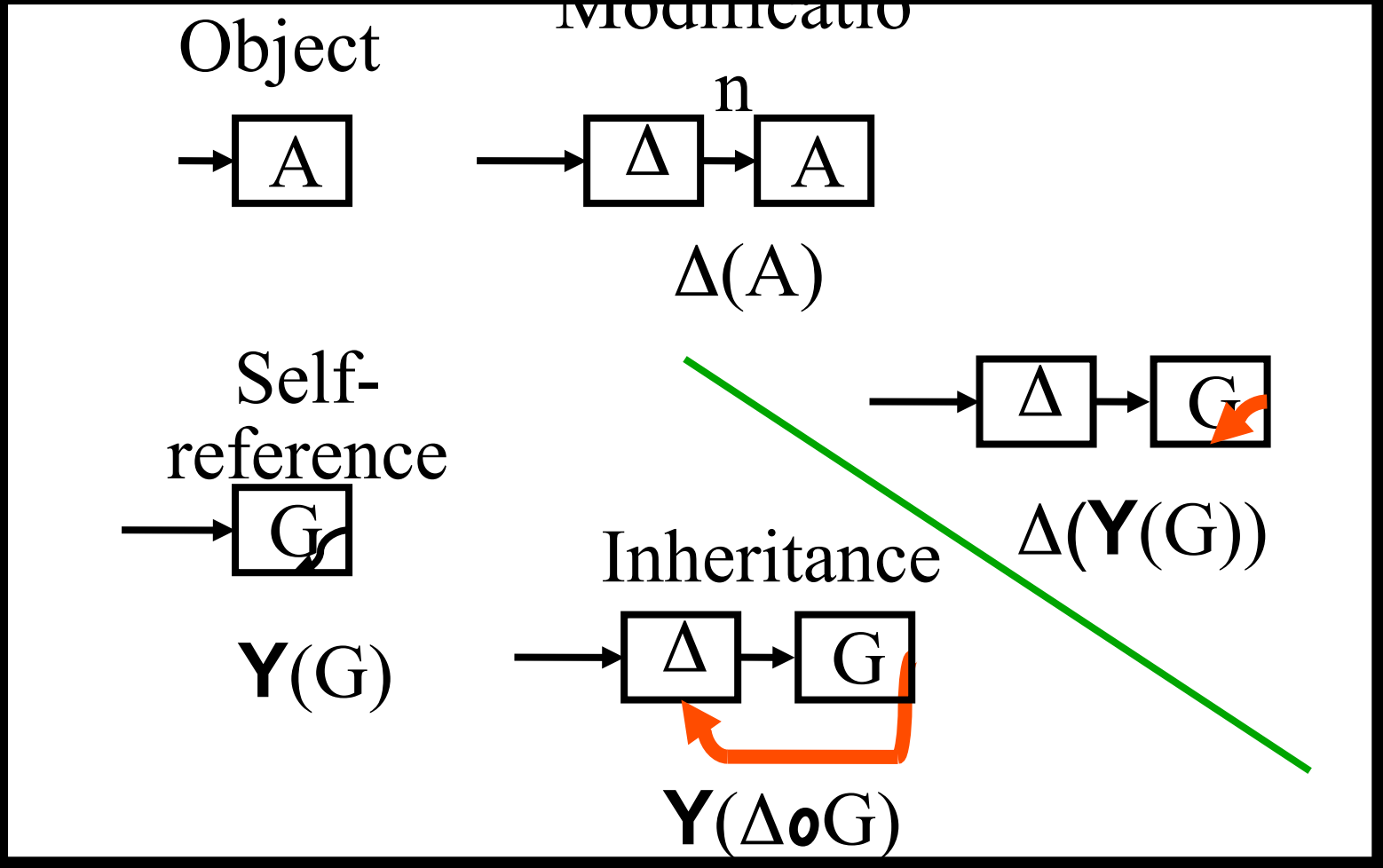
b

104

Inheritance
(in one slide)

105

Inheritance



106

History

107

1 User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction

John C. Reynolds

Abstract

User-defined types (or modes) and procedural (or functional) data structures are complementary methods for data abstraction, each providing a capability lacked by the other. With user-defined types, all information about the representation of a particular kind of data is centralized in a type definition and hidden from the rest of the program. With procedural data structures, each part of the program which creates data can specify its own representation, independently of any representations used elsewhere for the same kind of data. However, this decentralization of the description of data is achieved at the cost of prohibiting primitive operations from accessing the representations of more than one data item. The contrast between these approaches is illustrated by a simple example.

Introduction

User-defined types and procedural data structures have both been proposed as methods for data abstraction, i.e., for limiting and segregating the portion of a program that depends upon the representation used for some kind of data. In this paper we suggest, by means of a simple example, that these methods are complementary, each providing a capability lacked by the other.

The idea of user-defined types has been developed by Morris [1, 2], Liskov and Zilles [3], Fischer and Fischer [4], and Wulf [5], and has its roots in earlier work by Hoare and Dahl [6]. In this approach, each particular conceptual kind of data is called a *type*, and for each type used in a program, the program is divided into two parts: a type definition and an "outer" or "abstract" program. The type definition specifies the representation to be used for the data type and a set of primitive operations (and perhaps constants), each defined in terms of the representaton. The choice of representation is hidden from the outer program by requiring all manipulations of the data type in the outer program to be expressed in terms of the primitive operations. The heart of the matter is that any consistent change in the data representation can be effected by altering the type definition without changing the outer program.

Various notions of procedural (or functional) data structures have been developed by Reynolds [7], Landin [8], and Balzer [9]. In this approach, the abstract form of

First appeared in Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, 1975, IFIP Working Group 2.1 on Algol, INRIA, pages 157–168. Reprinted in David Gries, editor, *Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3*, 1978, Springer-Verlag, pages 309–317.

User-defined types and procedural data structures as complementary approaches to data abstraction

by J. C. Reynolds

New Advances in Algorithmic Languages INRIA,

108

Abstract data types

1 User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction

John C. Reynolds

Abstract

User-defined types (or modes) and procedural (or functional) data structures are complementary methods for data abstraction, each providing a capability lacked by the other. With user-defined types, all information about the representation of a particular kind of data is centralized in a type definition and hidden from the rest of the program. With procedural data structures, each part of the program which creates data can specify its own representation, independently of any representations used elsewhere for the same kind of data. However, this decentralization of the description of data is achieved at the cost of prohibiting primitive operations from accessing the representations of more than one data item. The contrast between these approaches is illustrated by a simple example.

Introduction

User-defined types and procedural data structures have both been proposed as methods for data abstraction, i.e., for limiting and segregating the portion of a program that depends upon the representation used for some kind of data. In this paper we suggest, by means of a simple example, that these methods are complementary, each providing a capability lacked by the other.

The idea of user-defined types has been developed by Morris [1, 2], Liskov and Zilles [3], Fischer and Fischer [4], and Wulf [5], and has its roots in earlier work by Hoare and Dahl [6]. In this approach, each particular conceptual kind of data is called a *type*, and for each type used in a program, the program is divided into two parts: a type definition and an "outer" or "abstract" program. The type definition specifies the representation to be used for the data type and a set of primitive operations (and perhaps constants), each defined in terms of the representaton. The choice of representation is hidden from the outer program by requiring all manipulations of the data type in the outer program to be expressed in terms of the primitive operations. The heart of the matter is that any consistent change in the data representation can be effected by altering the type definition without changing the outer program.

Various notions of procedural (or functional) data structures have been developed by Reynolds [7], Landin [8], and Balzer [9]. In this approach, the abstract form of

First appeared in Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, 1975, IFIP Working Group 2.1 on Algol, INRIA, pages 157–168. Reprinted in David Gries, editor, *Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3*, 1978, Springer-Verlag, pages 309–317.

~~User-defined types~~
and
~~procedural data structures~~ objects
as
complementary
approaches to
data abstraction

by J. C. Reynolds

New Advances in
Algorithmic Languages
INRIA, 1975

109

"[an object with two methods] is more a tour de force than a specimen of clear programming."

- J. Reynolds

110

Extensibility Problem (aka Expression Problem)

1975 Discovered by J. Reynolds
1990 Elaborated by W. Cook
1998 Renamed by P. Wadler
2005 Solved by M. Odersky (?)
2025 Widely understood (?)

Summary

112

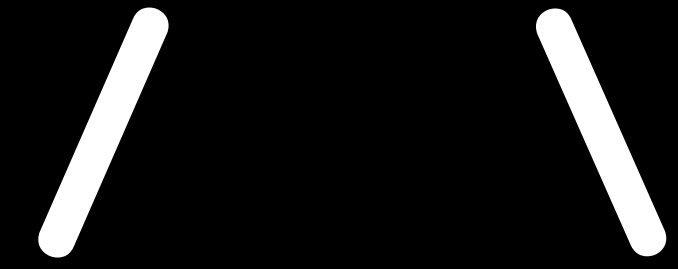
It is possible to do Object-Oriented programming in Java

113

Lambda-calculus was the first object-oriented language (1941)

114

Data Abstraction



ADT Objects

115