On Understanding Data Abstraction

...

Revisited

William R. Cook
The University
of Texas at Austin

Dedicated to P. Wegner

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Languages

Keywords object, class, abstract data type, ADT

1. Introduction

What is the relationship between *objects* and *abstract data types* (ADTs)? I have asked this question to many groups of computer scientists over the last 20 years. I usually ask it at dinner, or over drinks. The typical response is a variant of "objects are a kind of abstract data type".

This response is consistent with most programming language textbooks. Tucker and Noonan [57] write "A class is itself an abstract data type". Pratt and Zelkowitz [51] intermix discussion of Ada, C++, Java, and Smalltalk as if they were all slight variations on the same idea. Sebesta [54] writes "the abstract data types in object-oriented languages... are called classes." He uses "abstract data types" and "data abstraction" as synonyms. Scott [53] describes objects in detail, but does not mention abstract data types other than giving a reasonable discussion of opaque types.

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form. But syntactic blending does not erase fundamental semantic differences which affect flexibility, extensibility, safety and performance of programs. Therefore, to use modern programming languages effectively, one should understand the fundamental difference between objects and abstract data types.

While objects and ADTs are fundamentally different, they are both forms of *data abstraction*. The general concept of data abstraction refers to any mechanism for hiding the implementation details of data. The concept of data abstraction has existed long before the term "data abstraction" came into existence. In mathematics, there is a long history of abstract representations for data. As a simple example, consider the representation of integer sets. Two standard approaches to describe sets abstractly are as an *algebra* or as a *characteristic function*. An algebra has a sort, or collection of abstract values, and operations to manipulate the values[1]. The characteristic function for a set maps a domain of values to a boolean value, which indicates whether or not the value is included in the set. These two traditions in mathematics correspond closely to the two forms of data abstraction in programming: algebras relate to abstract data types, while characteristic functions are a form of object.

In the rest of this essay, I elaborate on this example to explain the differences between objects and ADTs. The

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

[1] The sort, or carrier set, of an algebra is often described as a set, making this definition circular. Our goal is to define specific set abstractions with restricted operations, which may be based on and assume a more general concept of sets

Objects

????

Abstract Data Types

Warnings!

No “Objects Model the Real World”

No Inheritance

No Mutable State

No Subtyping!

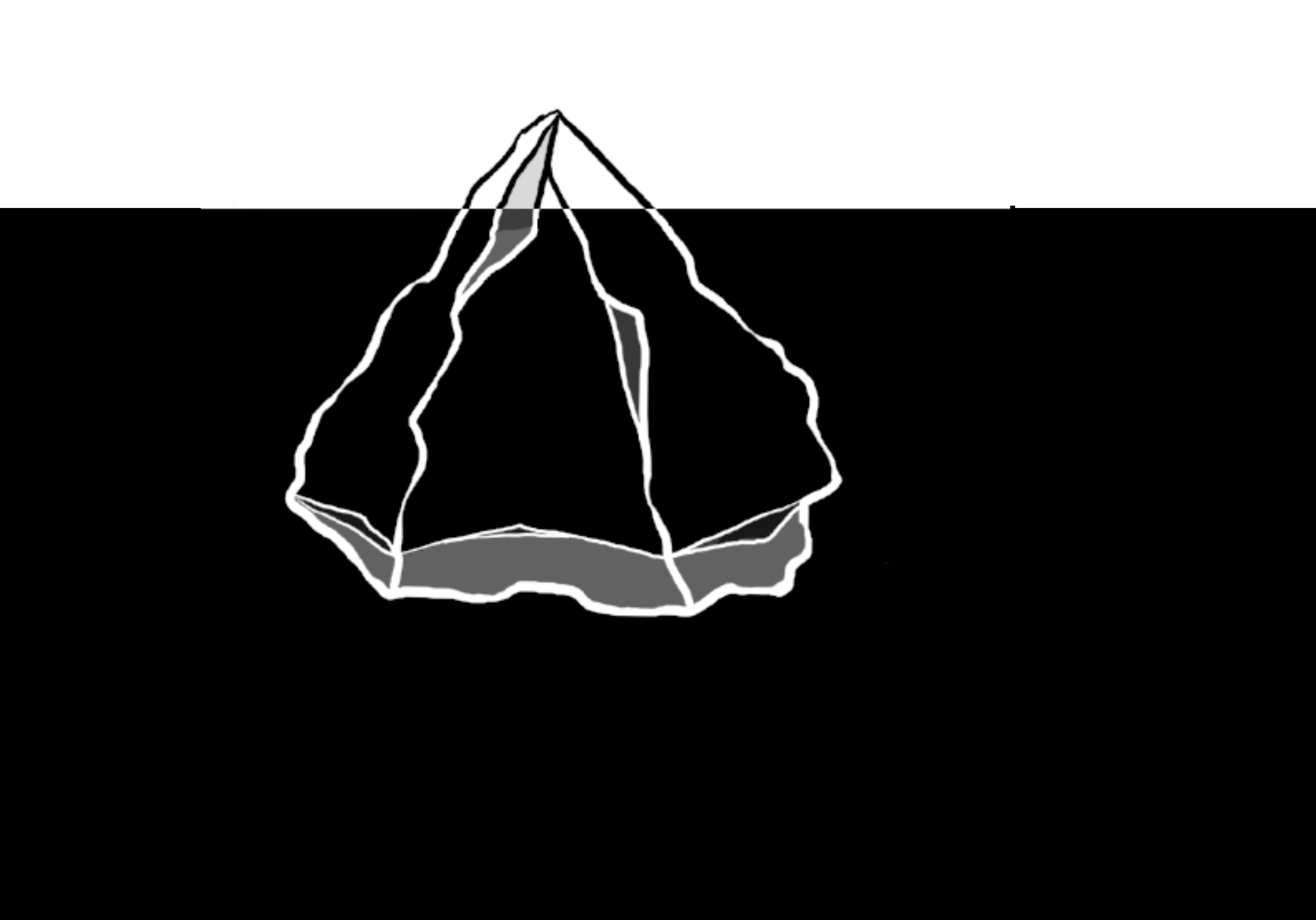
Interfaces as types

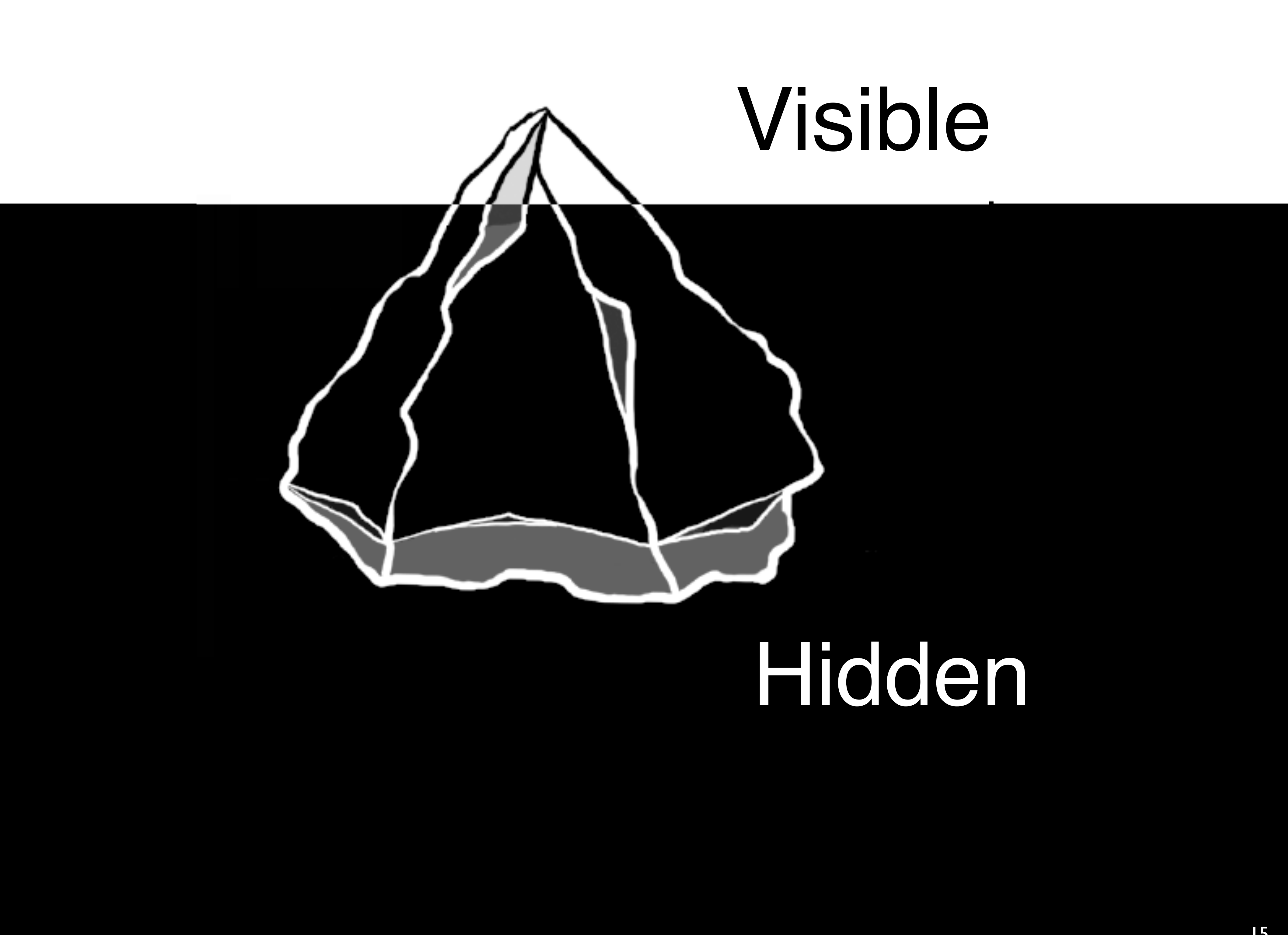
Not Essential

(very nice but not essential)

[discuss inheritance later]

Abstraction




Visible
Hidden

Procedural Abstraction

```
bool f(int x) { … }
```

Procedural Abstraction

int → bool

(one kind of)

Type Abstraction

$\forall T.Set[T]$

Abstract Data Type

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

Abstract Data Type

Abstract

signature Set

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

Type
+
Operations

ADT Implementation

abstype Set = List of Int

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

Using ADT values

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

Visible name: Set

Hidden representation: List of Int

ISetModule = $\exists$Set.{

```
empty       : Set
insert      : Set, Int → Set

isEmpty     : Set → Bool
contains    : Set, Int → Bool
```

}

Natural!

just like
built-in types

Mathematical
Abstract Algebra

Type Theory

$\exists x.P$

(existential types)

Abstract Data Type

=

Data Abstraction

Right?

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

Another way

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

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

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

Sets as characteristic functions

type Set =

Int → Bool

Empty =

λn. false

$$\text{Insert}(s, m) =$$

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

Using them is easy

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

So What?

Flexibility

set of all
even numbers

Set ADT:
Not Allowed!

or…
break open ADT
& change
representation

set of even numbers as a function?

Even =

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

Even interoperates

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

Sets-as-functions are objects!

No type abstraction required!

$$\text{type Set} = \text{Int} \rightarrow \text{Bool}$$

multiple methods?

sure...

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

What about Empty and Insert?

(they are classes)

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

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

Using Classes

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

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

Including more methods

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

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

Type Recursion

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

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

Value Recursion

Using objects

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

Autognosis

Autognosis

(Self-knowledge)

Autognosis

An object can access other objects only through public interfaces

operations
on
multiple objects?

union of two sets

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

  ...
}
```

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

Complex Operation

(binary)

intersection

of

two sets

??

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

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

Autognosis:
Prevents some
operations
(complex ops)

Autognosis: Prevents some optimizations (complex ops)

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

Objects are fundamentally different from ADTs

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
}

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'')

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'')

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'')

Objects are fundamental (too)

Mathematical

functional
representation
of data

Type Theory

$\mu X.P$

(recursive types)

ADTs <u>require</u>
a
static type system

Objects work well with or without static typing

“Binary” Operations?

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

Objects are very higher-order

(functions passed as data and returned as results)

Verification

ADTs: construction

Objects: observation

ADTs: induction

Objects: coinduction
complicated by: callbacks,
state

Objects are designed to be as difficult as possible to verify

Simulation

One object can simulate another! (identity is bad)

Java

What is a type?

Declare variables

Classify values

Class as type

=> representation

Class as type

=> ADT

Interfaces as type

=> behavior
pure objects

Harmful!

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

Object-Oriented subset of Java: class name used <u>only</u> after “new”

It’s not an accident that “int” is an ADT in Java

Smalltalk

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

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

True =

λ a . λ b .

a

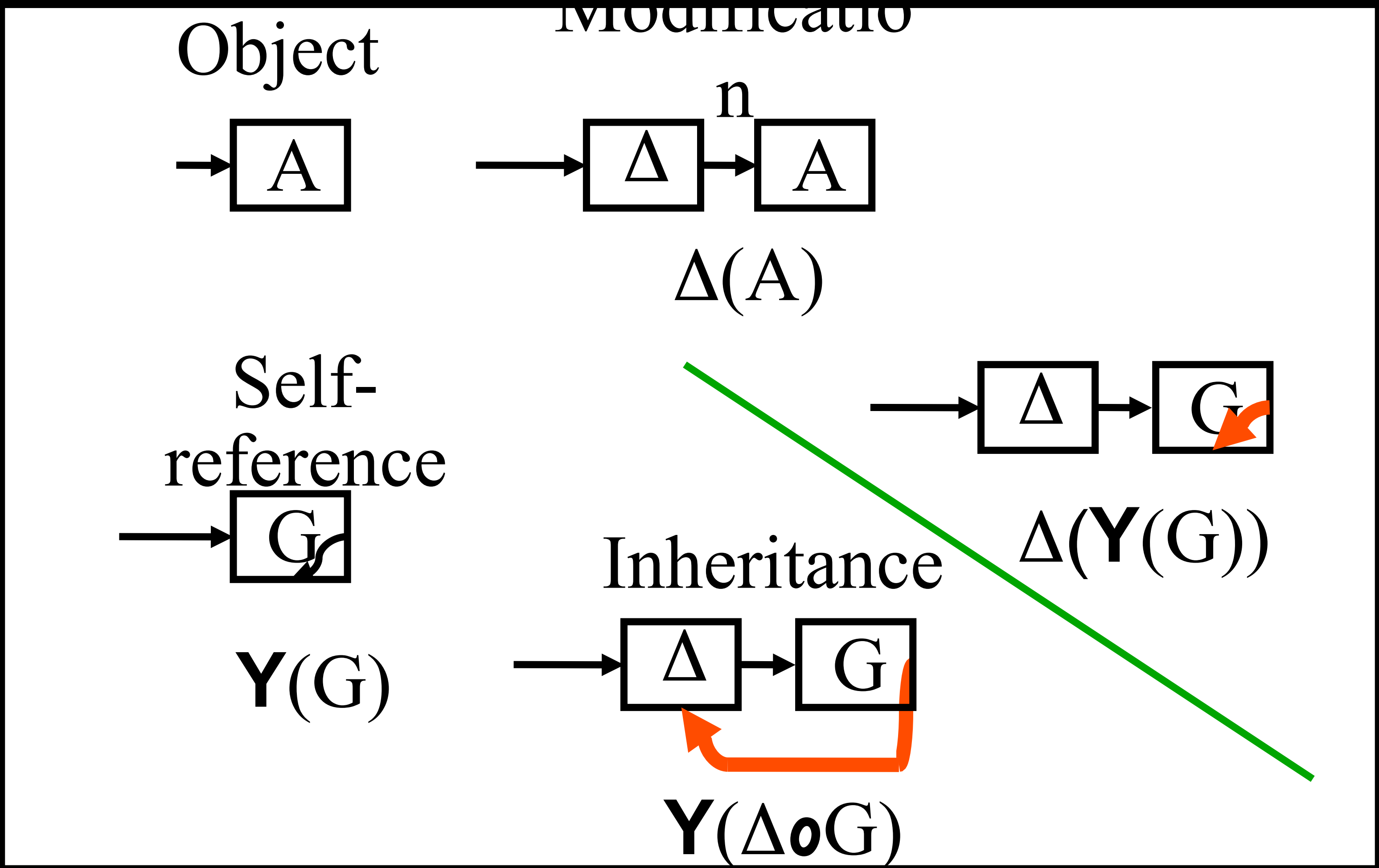
False =

λ a . λ b .

b

Inheritance
(in one slide)

Inheritance



History

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,

Abstract data types

~~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

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. © Springer-Verlag, attn: Permissions Dept., 175 Fifth Ave, 19 Flr, New York, NY 10010.

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

- J. Reynolds

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

It is possible to do Object-Oriented programming in Java

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

Data Abstraction

/ \

ADT Objects