On Understanding Data Abstraction Revisited
William R. Cook
The University of Texas at Austin

Dedicated to P. Wegner
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
Procedural Abstraction

bool f(int x) { ... }
Procedural Abstraction

int → bool
(one kind of)
Type
Abstraction

\( \forall T. \text{Set}[T] \)
Abstract Data Type

signature Set
empty : Set
insert : Set, Int \rightarrow Set
is EMPTY : Set \rightarrow Bool
contains : Set, Int \rightarrow Bool
Abstract Data Type

signature Set
empty : Set
insert : Set, Int → Set
isEmpty : Set → Bool
contains : Set, Int → Bool
Type
+
Operations
ADT Implementation

\[
\begin{align*}
\text{abstype Set} & \ = \ \text{List of Int} \\
\text{empty} & \ = \ [] \\
\text{insert}(s, n) & \ = \ (n : s) \\
\text{isEmpty}(s) & \ = \ (s == []) \\
\text{contains}(s, n) & \ = \ (n \in s)
\end{align*}
\]
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 = ∃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

∃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 =

\[ \lambda n. \text{false} \]
\begin{align*}
\text{Insert}(s, m) &= \\
\lambda n. \ (n= m) \lor s(n)
\end{align*}
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 =

\[ \lambda n. \ (n \ mod \ 2 = 0) \]
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!

```
type Set = Int → Bool
```
multiple methods?
sure...
interface Set {
    contains: Int \rightarrow 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 \to Bool
    isEmpty : Bool
    insert : Int \to 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)

\[
\text{Set} = \{ \\
\quad \text{isEmpty} : \text{Bool} \\
\quad \text{contains} : \text{Int} \rightarrow \text{Bool} \\
\quad \text{insert} : \text{Int} \rightarrow \text{Set} \\
\quad \text{union} : \text{Set} \rightarrow \text{Set} \\
\}
\]

\[
\text{Empty} : \text{Set} \\
\text{Insert} : \text{Set} \times \text{Int} \rightarrow \text{Set} \\
\text{Union} : \text{Set} \times \text{Set} \rightarrow \text{Set}
\]

ADT  
(existential types)

\[
\text{SetImpl} = \exists \text{Set} . \{ \\
\quad \text{empty} : \text{Set} \\
\quad \text{isEmpty} : \text{Set} \rightarrow \text{Bool} \\
\quad \text{contains} : \text{Set}, \text{Int} \rightarrow \text{Bool} \\
\quad \text{insert} : \text{Set}, \text{Int} \rightarrow \text{Set} \\
\quad \text{union} : \text{Set}, \text{Set} \rightarrow \text{Set} \\
\}
\]
## Operations/Observations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty(s)</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>contains(s, n)</td>
<td>false</td>
<td>$n = m \lor\ \text{contains}(s', n)$</td>
</tr>
<tr>
<td>insert(s, n)</td>
<td>false</td>
<td>Insert(s, n)</td>
</tr>
<tr>
<td>union(s, s'')</td>
<td>isEmpty(s'')</td>
<td>Union(s, s'')</td>
</tr>
</tbody>
</table>
### ADT Organization

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>Insert(s', m)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>isEmpty(s)</strong></td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td><strong>contains(s, n)</strong></td>
<td>false</td>
<td>n=m ∨ contains(s', n)</td>
</tr>
<tr>
<td><strong>insert(s, n)</strong></td>
<td>false</td>
<td>Insert(s, n)</td>
</tr>
<tr>
<td><strong>union(s, s'')</strong></td>
<td>isEmpty(s'')</td>
<td>Union(s, s'')</td>
</tr>
</tbody>
</table>
## 00 Organization

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>s''</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty(s)</td>
<td>Empty true</td>
<td>Insert(s', m) false</td>
</tr>
<tr>
<td>contains(s, n)</td>
<td>false</td>
<td>n=m ∨ contains(s', n)</td>
</tr>
<tr>
<td>insert(s, n)</td>
<td>false</td>
<td>Insert(s, n)</td>
</tr>
<tr>
<td>union(s, s'')</td>
<td>isEmpty(s'')</td>
<td>Union(s, s'')</td>
</tr>
</tbody>
</table>
Objects are fundamental (too)
Mathematical functional representation of data
Type Theory

$\mu \times . P$

(recursive types)
ADTs require 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 only 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 =
\[ \lambda a . \lambda b . a \]

False =
\[ \lambda a . \lambda b . b \]
Inheritance
(in one slide)
Inheritance

Object

\[ \Delta \]

\[ \Delta(A) \]

Self-reference

\[ \Delta \]

\[ \Delta(Y(G)) \]

Δ(Y(Δ₀G))

Inheritance

Δ(Y(G))

Δ(A)

Δ(Y(G))

Δ(A)

Δ(Y(G))

Δ(A)

Δ(Y(G))
History
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 and procedural data structures have both been proposed as complementary approaches to data abstraction. User-defined types and procedural data structures are two ways of defining data abstractions. User-defined types are defined by the programmer, while procedural data structures are defined by the system. The advantage of user-defined types is that they are more flexible and can be used in a wider range of situations. However, they are also more complex and require more effort to implement.

Procedural data structures, on the other hand, are easier to implement but more limited in their expressiveness. They are defined in terms of operations that can be performed on the data, rather than in terms of the data itself. This makes them more restricted, but it also makes them easier to understand and use.

Both approaches have their strengths and weaknesses, and the choice between them depends on the specific requirements of the program. In some cases, it may be possible to use both approaches, and the program can be designed to work with both user-defined types and procedural data structures.

By combining the strengths of both approaches, it is possible to create a more powerful and flexible data abstraction layer that can be used in a wide range of situations. This allows programmers to write more efficient and maintainable code, and it also allows them to take advantage of the strengths of both user-defined types and procedural data structures.
“[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