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

Dedicated to P. Wegner
Objects

?????

Abstract Data Types
Ignore
non-essentials:
“Objects Model the Real World”
Inheritance
Mutable State
Subtyping
... these are not essential for OOP

(very nice but not essential)
Essential: Interfaces as types
discuss inheritance later
Abstraction
bool f(int x) { ... }
Procedural Abstraction

int → bool
(one kind of)

Type

Abstraction

class Set<T>
(one kind of)
Type
Abstraction
∀T. Set[T]
Abstract Data Type

signature Set

empty : Set
insert : Set, Int → Set
is EMPTY : Set → Bool
contains : Set, Int → Bool
Abstract Data Type

signature \text{Set} \\
empty : \text{Set} \\
insert : \text{Set}, \text{Int} \rightarrow \text{Set} \\
isEmpty : \text{Set} \rightarrow \text{Bool} \\
contains : \text{Set}, \text{Int} \rightarrow \text{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

Set x = empty
Set y = insert(x, 3)
Set z = insert(y, 5)
print( contains(z, 2) )

==> false
Hidden representation: List of Int

Visible name: Set
I\text{SetModule} = \exists \text{Set}. \{
\text{empty} : \text{Set} \\
\text{insert} : \text{Set}, \text{Int} \rightarrow \text{Set} \\
\text{isEmpty} : \text{Set} \rightarrow \text{Bool} \\
\text{contains} : \text{Set}, \text{Int} \rightarrow \text{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 =

\( \lambda n. \text{false} \)
Insert(s, m) =

\( \lambda n. \ (n == m) \) or s(n)
Using them is easy

Set x = Empty
Set y = Insert(x, 3)
Set z = 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 =

\[ \lambda n. \ (n \ % \ 2 \ ==\ 0) \]
Even interoperates

Set x = Even
Set y = Insert(x, 3)
Set z = Insert(y, 5)
print( z(2) )
==> true
Sets-as-functions are objects!
No type abstraction

```haskell
type Set = Int → 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)
        or s.contains(n); }
    isEmpty() { return false; }
}
Using Classes

Set x = Empty()
Set y = Insert(x, 3)
Set z = 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

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

An object can only access other objects through public interfaces
operations on multiple objects?
union of two sets
class Union(a, b) {
    contains(n) { a.contains(n)
        or b.contains(n); }
    isEmpty() { a.isEmpty(n)
        and 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) and b.contains(n); }
    isEmpty() { ???no way!??? }
    ...
}

Autognosis:
complicates some
operations
(complex ops)
Autognosis:
complicates some optimizations
(complex ops)
Inspecting two representations & optimization is easy in ADT
Objects are fundamentally different from ADTs.
Object Interface
(recurisve types)

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

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty(s)</td>
<td>Checks if the set <code>s</code> is empty. Returns <code>true</code> if empty, <code>false</code> otherwise.</td>
</tr>
<tr>
<td>contains(s, n)</td>
<td>Checks if <code>n</code> is contained in <code>s</code>. Returns <code>true</code> if contained, <code>false</code> otherwise.</td>
</tr>
<tr>
<td>insert(s, n)</td>
<td>Inserts <code>n</code> into <code>s</code>.</td>
</tr>
<tr>
<td>union(s, s'')</td>
<td>Performs union operation between <code>s</code> and <code>s''</code>.</td>
</tr>
</tbody>
</table>

### Table:

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

\[ \mu X. P \]

(recursive types)
ADTs require a static type system
Objects work great with dynamic 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 is only after “new”
Its not an accident that "int" is an ADT in Java
Smalltalk
class True
  ifTrue: a ifFalse: b
  ^a value

class False
  ifTrue: a ifFalse: b
  ^b value
True = 
\( \lambda a. \lambda b. \ a \)

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

Object

A
Inheritance

Object  Modification

A  \rightarrow \Delta \rightarrow \Delta(A)
Inheritance

Object

\[ A \]

\[ \rightarrow \]

Modification

\[ \Delta \rightarrow A \]

\[ \Delta(A) \]

Self-reference

\[ G \]

\[ \rightarrow \]

\[ Y(G) \]
Inheritance

Object

\[ \Delta \rightarrow A \]

Modification

\[ \Delta(A) \]

Self-reference

\[ \Delta \rightarrow G \]

\[ \Delta(Y(G)) \]
Inheritance
History
User-defined types and procedural data structures as complementary approaches to data abstraction

by J. C. Reynolds

New Advances in Algorithmic Languages
INRIA, 1975
Abstract data types

User-defined types

and

objects

procedural data structures

as

complementary approaches

to

data abstraction

by J. C. Reynolds

New Advances in Algorithmic Languages

INRIA, 1975
"[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