On Understanding Data Abstraction

. . .

Revisited

William R. Cook
The University
of Texas at Austin

Dedicated to P. Wegner

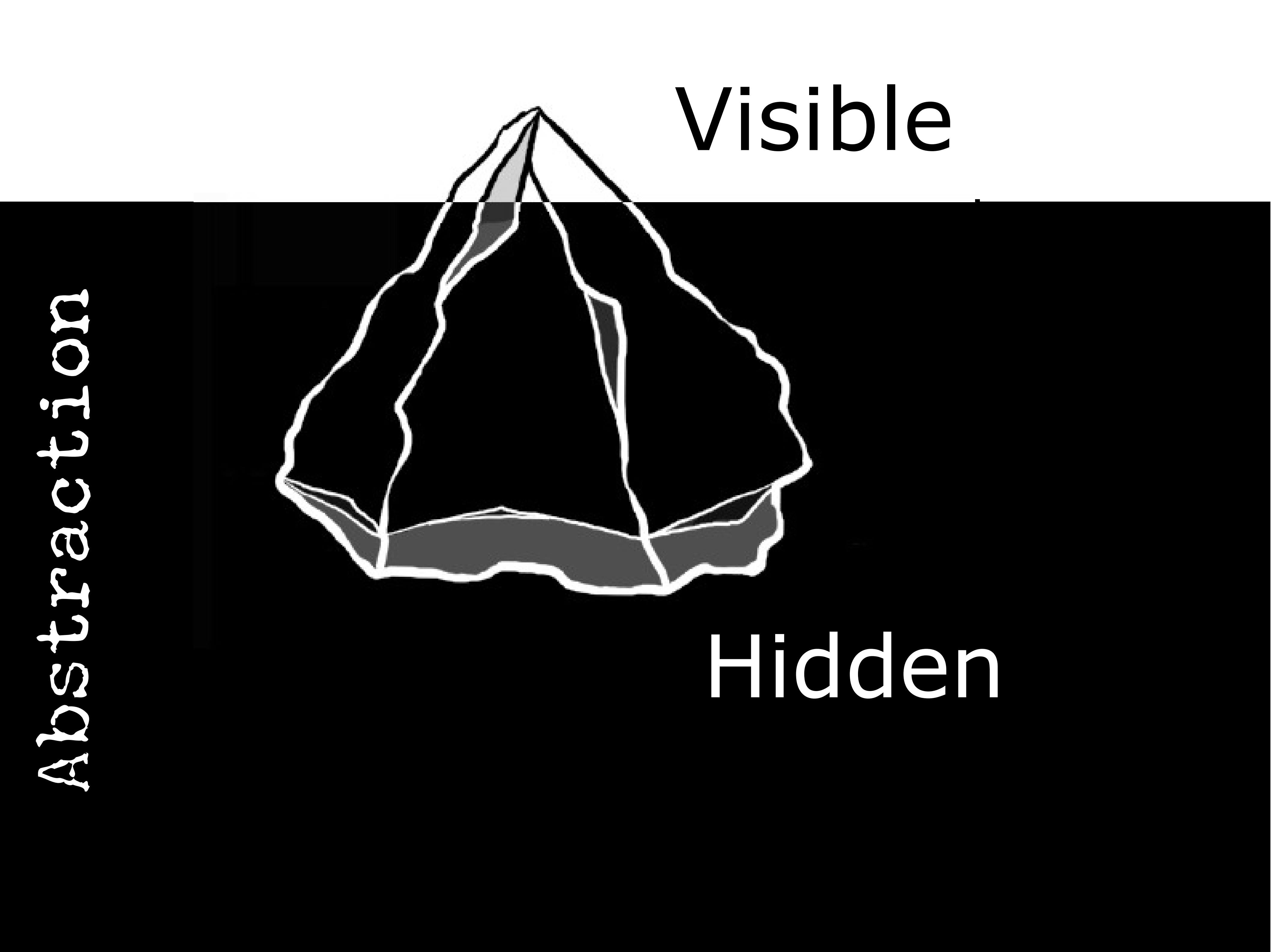
Objects

....

Abstract Data Types

Non-essentials:

Inheritance
Mutable State
Subtyping


Visible
Hidden
Abstraction

Procedural Abstraction

bool f(int x) { … }

Procedural Abstraction

$int \rightarrow bool$

(one kind of)

Type Abstraction

class Set<T>

(another kind of)

Type

Abstraction

$\exists T.Set[T]$

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

Hidden Type

+

Operations

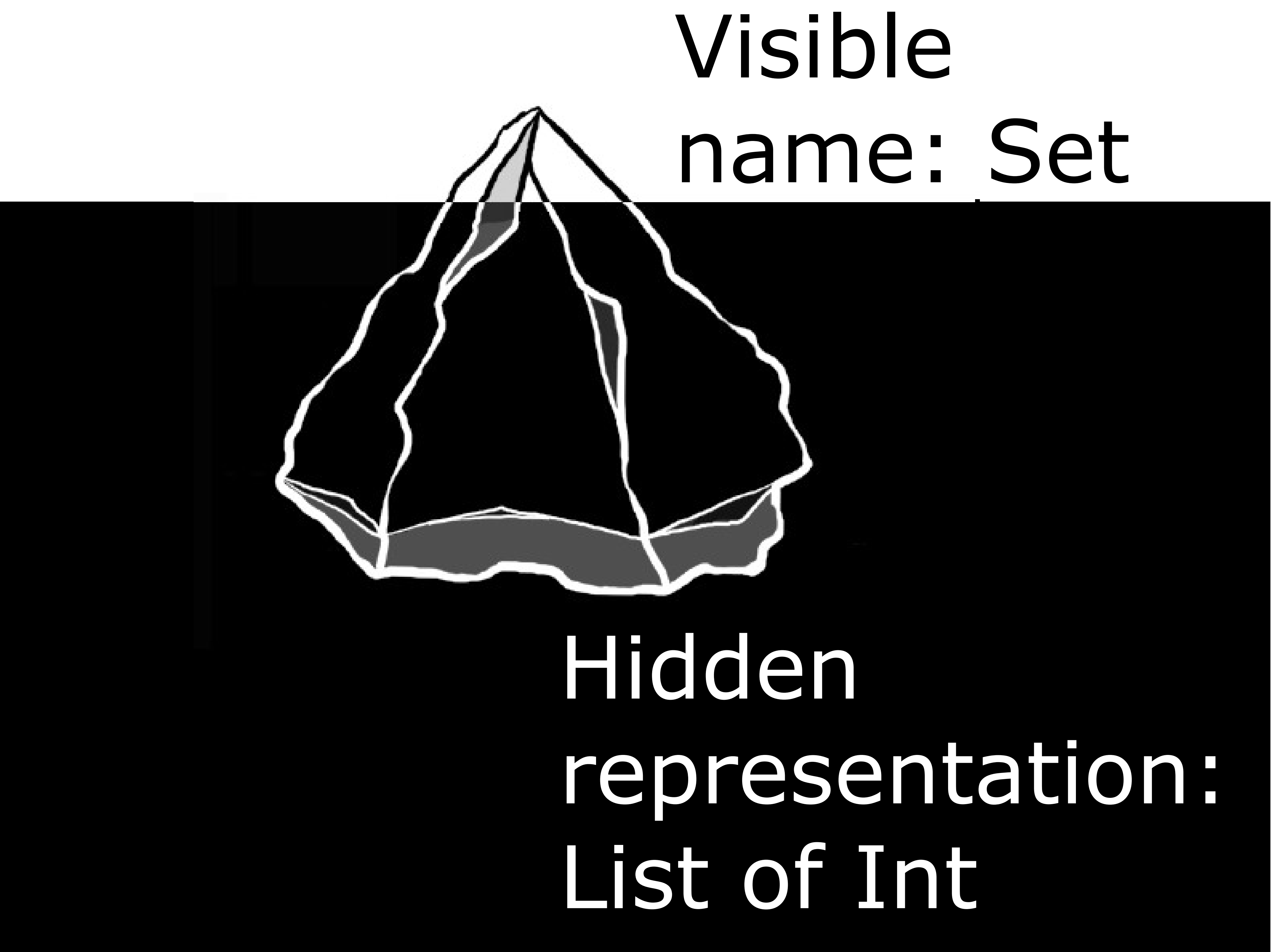
```
struct  FILE;

FILE* fopen(char *, char *);
int     feof(FILE*);
int     fgetc(FILE*);
int     fputc(int, FILE*);
char* fgets(char *, int, FILE*);
int     fputs(char *, FILE*);
int     fclose(FILE*);
```

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
```


Visible name: Set
Hidden representation: List of Int

```
ISetImpl = ∃Set.{
    empty    : Set
    insert   : Set, Int → Set

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

Natural!

Just like
built-in types
(int, bool, etc)

Mathematical!

Abstract Algebra

Theoretical!

$$\exists t.T$$

(existential types)

Data Abstraction

=

Abstract Data Type

Right?

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

Another way

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

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

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

Sets as characteristic functions

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

Empty =

λn. false

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

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

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

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

multiple

methods?

sure...

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

What about

Empty and Insert?

(they are classes)

```
Empty = record {
  contains= λn. false;
  isEmpty=  true;
}
```

```
Insert(s, m) = record {
  contains= λn.(n=m)
                  or s.contains(n);
  isEmpty=  false;
}
```

Using Classes

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

An object

<u>is</u>

the 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

```
Empty = record {
  contains= λn. false;
  isEmpty=  true;
  insert= λn. Insert(this, n)
}
```

```
Empty = μthis. record {
  contains= λn. false;
  isEmpty=  true;
  insert= λn. 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

An object can only access other objects through public interfaces operations

on

multiple objects?

union

of

two sets

```
Union(a, b) = record {
  contains= λn. a.contains(n)
                  or b.contains(n);
  isEmpty= a.isEmpty()
                 and b.isEmpty();
  ...
}
```

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

Complex Operation
(binary)

intersection

of

two sets

??

```
Intersection(a, b) = record {
  contains= λn. a.contains(n)
                and b.contains(n);

  isEmpty =  ???no way!???
  ...
}
```

Autognosis:
complicates some
operations
(complex ops)

Inspecting two representations & optimization is easy in ADT

Objects are

encapsulated

from

each other

Object Interface
(recursive types)

```
Set = {
  isEmpty  : Bool
  contains : Int → Bool
  insert   : Int → Set
  union    : Set → Set
}
Empty  : Set
Insert : Set, Int → Set
Union  : Set, 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)	Insert(s, n)	Insert(s, n)
union(s, s'')	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)	Insert(s, n)	Insert(s, n)
union(s, s'')	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)	Insert(s, n)	Insert(s, n)
union(s, s'')	s''	Union(s, s'')

(no pattern matching)

Objects are fundamental (too)

Mathematical!

functional

representation

of data

Theoretical!

$\mu t.T$

(recursive types)

Natural!

Machines with hidden behavior

ADTs require

a

static type system

Objects work great with dynamic typing

“Binary” Operations?

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

Objects are

<u>very</u>

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

COM

is a pure

OO system

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

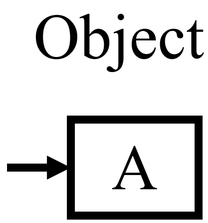
Inheritance

not necessary for OO

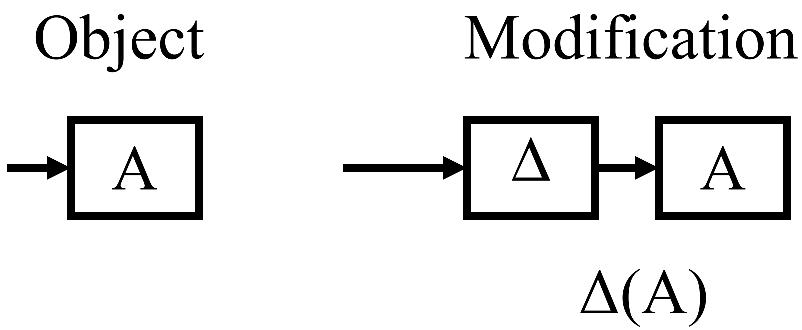
not specific to OO

but fundamentally *new!*

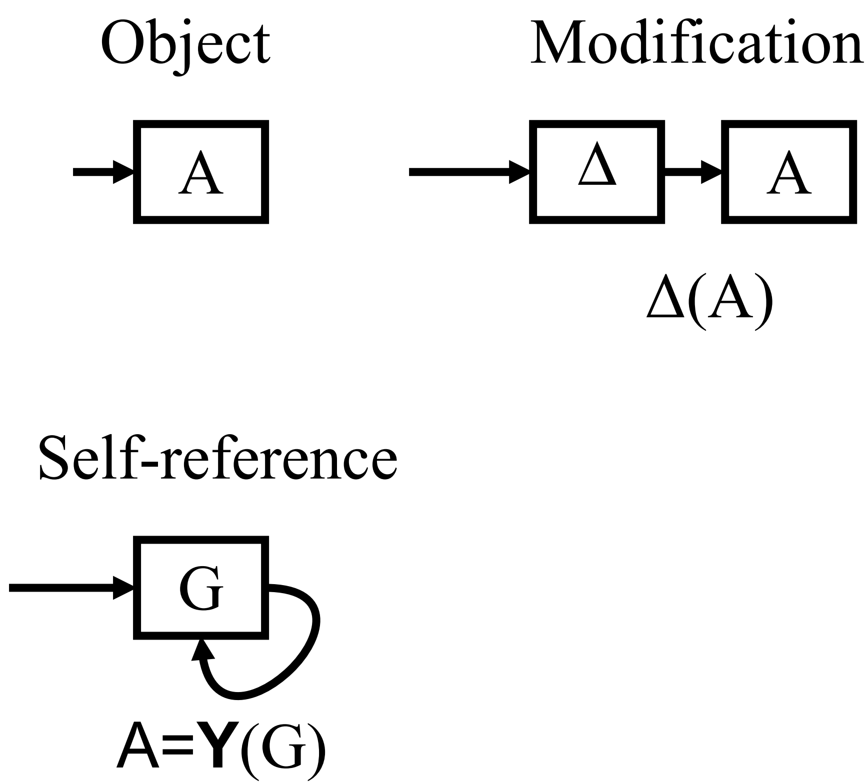
Inheritance



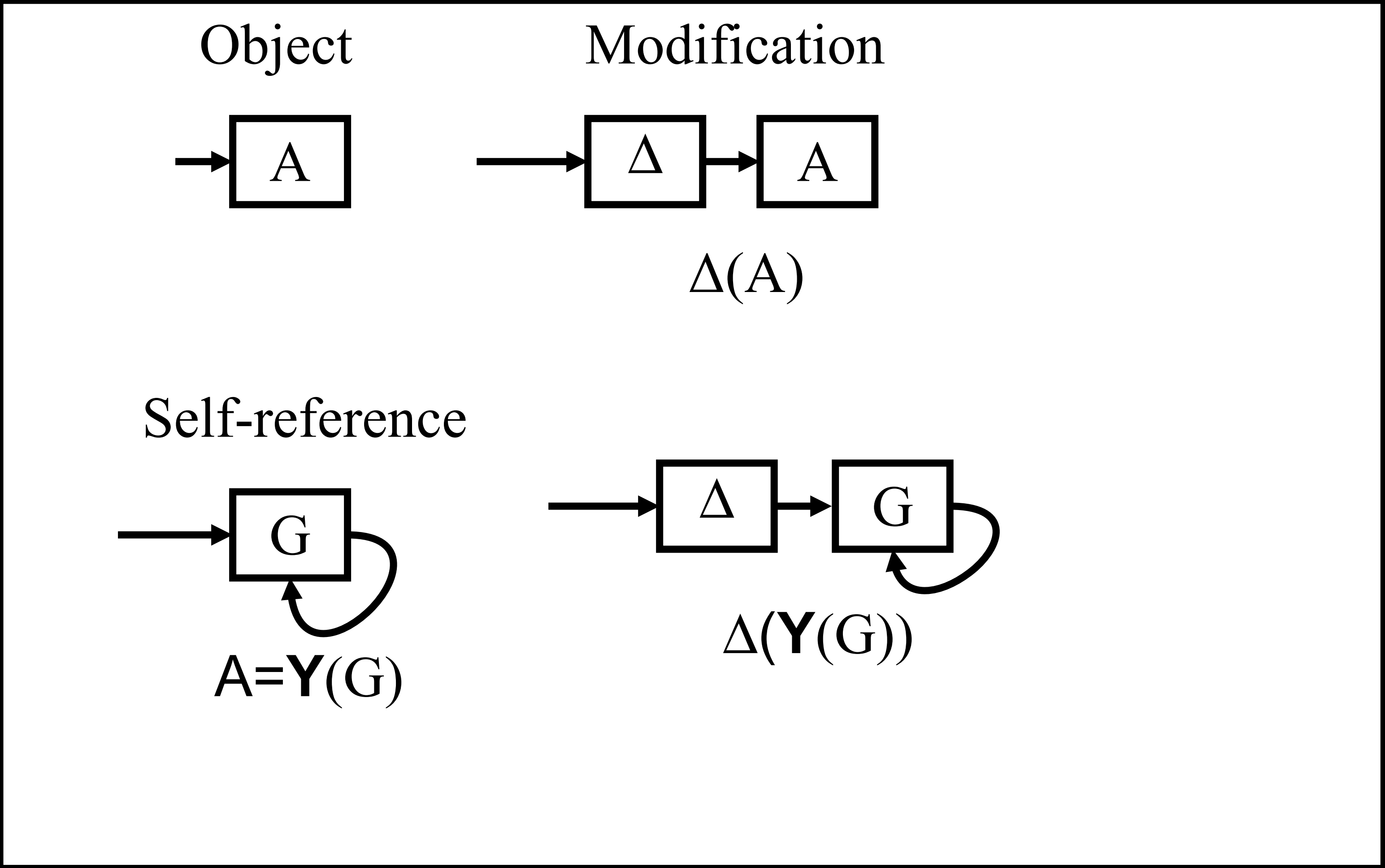
Inheritance



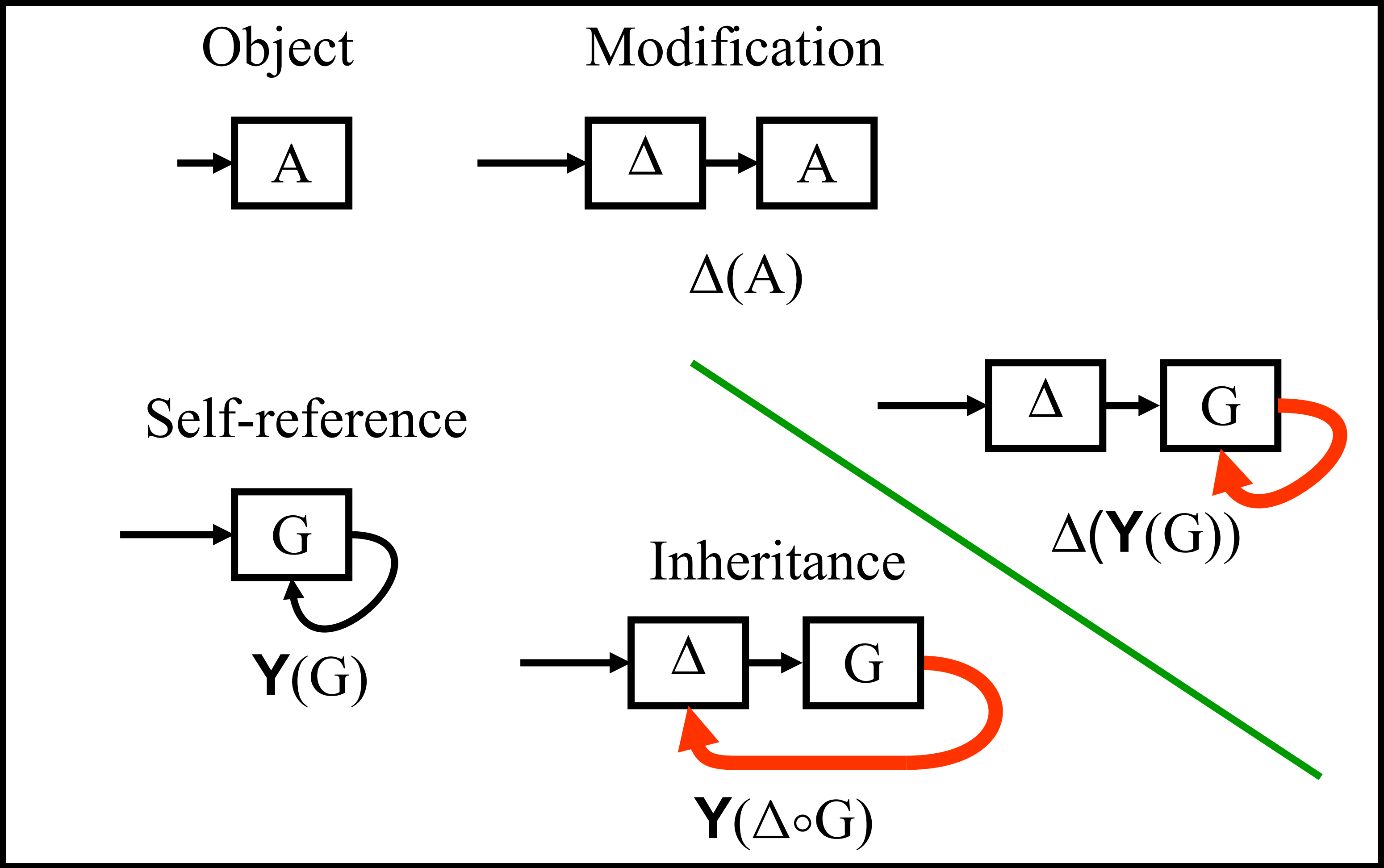
Inheritance



Inheritance



Inheritance



Inheritance =

Incremental Programming

+

Self-reference

History

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 (?)

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

Summary

Not all data

is

Sums of Products

Objects

are

Procedural

Data Abstractions

It is possible to do Object-Oriented programming in Java/C#

Operational semantics obscures meaning

Data Abstraction

/ \

ADT Objects