

# LINFO1104 – LSINC1104

## Concepts, paradigms, and semantics of programming languages

### Lecture 6 & 7

Peter Van Roy

ICTEAM Institute  
Université catholique de Louvain

[peter.vanroy@uclouvain.be](mailto:peter.vanroy@uclouvain.be)



1

## Overview of lectures 6 and 7

- Mutable state
  - So far we have done pure functional programming
  - Functional programming is modeled by the lambda calculus and has strong mathematical properties
  - But it's not enough! The same thing that makes it powerful (functions cannot change) is a weakness (sometimes functions must change).
- Data abstraction
  - Data abstraction is the main organizing principle for building complex software systems
  - Data abstraction is built upon two concepts: higher-order programming and mutable state



2

# Time and change in programs



3

## Time and change



- In functional programming, **there is no notion of time**
  - All functions are mathematical functions; once defined they never change
  - Programs do execute on a real machine, but a program cannot observe the execution of another program or of part of itself
    - It can only see the results of a function call, not the execution itself
    - Observing an execution of a program can only be done outside of the program's implementation
- In the real world, **there is time and change**
  - Organisms change their behavior over time, they grow and learn
  - How can we model this in a program?
- We need to add **time** to a program
  - Time is a complicated concept! Let us start with a simplified version of time, an **abstract time**, that keeps the essential property that we need: **modeling change**.

4



## State as an abstract time (1)

- Here's one solution: We define the abstract time as a sequence of values and we call it a state
- A **state** is a sequence of values calculated progressively, which contains the intermediate results of a computation
- The functional paradigm can use state according to this definition!
- The definition of Sum given here has a state

```
fun {Sum Xs A}
  case Xs
  of nil then A
  [] X|Xr then
    {Sum Xr A+X}
  end
end

{Browse {Sum [1 2 3 4] 0}}
```

5



## State as an abstract time (2)

- The two arguments Xs and A give us an **implicit state**
- | Xs        | A  |
|-----------|----|
| [1 2 3 4] | 0  |
| [2 3 4]   | 1  |
| [3 4]     | 3  |
| [4]       | 6  |
| nil       | 10 |
- It is **implicit** because the language has not changed
    - It is purely in the programmer's head: the programmer observes the changes in the program
  - In most cases this is not good enough: **we want the program itself to observe the changes**
    - We need a language extension!
    - We leave the functional paradigm and enter another paradigm

```
fun {Sum Xs A}
  case Xs
  of nil then A
  [] X|Xr then
    {Sum Xr A+X}
  end
end

{Browse {Sum [1 2 3 4] 0}}
```

6

# Explicit state (mutable state)




7

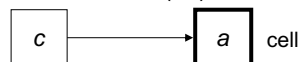
## Adding mutable state to the language



- We can make the state explicit by **extending the language**
- With this extension a program can **directly observe** the sequence of values in time
  - This was not possible in the functional paradigm
- We call our extension a **cell**
  - The word "cell" is chosen to avoid confusion with related terms, such as the overused word "variable"
- A cell is a **box with a content**
  - The content can be changed but the box remains the same
  - The same cell can have different contents: we can observe change
  - The sequence of contents is a state

 An unbound variable

Creating a cell with initial content  $a$  ( $=5$ )



Replace the content by another variable  $b$  ( $=6$ )




8



## A cell




- A cell is a box with **an identity** and **a content**
  - The identity is a constant (the “name” or “address” of the cell)
  - The content is a variable (in the single-assignment store)
- The content can be replaced by another variable

 An unbound variable

Creating a cell with initial content *a* (=5)

 →  cell

Replace the content by another variable *b* (=6)

 →  cell

```
A=5
B=6
C={NewCell A} % Create a cell
{Browse @C}   % Display content
C:=B          % Change content
{Browse @C}   % Display content
```

9

## Adding cells to the kernel language

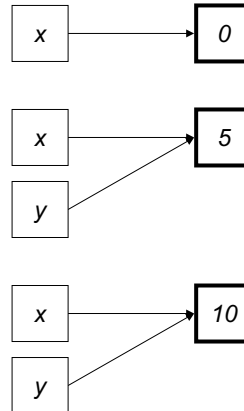


- We add cells and their operations
  - Cells have three operations
- **C={NewCell A}**
  - Create a new cell with initial content A
  - Bind C to the cell's identity
- **C:=B**
  - Check that C is bound to a cell's identity
  - Replace the cell's content by B
- **Z=@C**
  - Check that C is bound to a cell's identity
  - Bind Z to the cell's content

10

## Some examples (1)

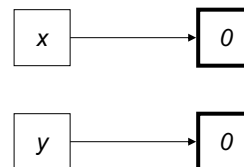
- `X={NewCell 0}`
- `X:=5`
- `Y=X`
- `Y:=10`
- `@X==10 % true`
- `X==Y % true`



11

## Some examples (2)

- `X={NewCell 0}`
- `Y={NewCell 0}`
- `X==Y % false`
- Because X and Y refer to different cells, with different identities
- `@X==@Y % true`
- Because the contents of X and Y are the same value



12

# Semantics of cells



13

## Semantics of cells (1)



- We have extended the kernel language with cells
  - Let us now extend the abstract machine to explain how cells execute
- There are now **two stores** in the abstract machine:
  - **Single-assignment store** (contains **variables**: immutable store)
  - **Multiple-assignment store** (contains **cells**: mutable store)
- A cell is a **pair** of two variables
  - The first variable is bound to the name of the cell (a constant)
  - The second variable is the cell's content
- Assigning a cell to a new content
  - **The pair is changed**: the second variable in the pair is replaced by another variable (the first variable stays the same)



**Warning: The variables do *not* change!** The single-assignment store is unchanged when a cell is assigned.

14

## Semantics of cells (2)



- The full store  $\sigma = \sigma_1 \cup \sigma_2$  has two parts:
  - **Single-assignment store** (contains **variables**)  
 $\sigma_1 = \{t, u, v, x=\xi, y=\zeta, z=10, w=5\}$
  - **Multiple-assignment store** (contains **pairs**)  
 $\sigma_2 = \{x:t, y:w\}$
- In  $\sigma_2$  there are two cells,  $x$  and  $y$ 
  - The name of  $x$  is the constant  $\xi$ , the name of  $y$  is  $\zeta$
  - The operation  $X:=Z$  changes  $x:t$  into  $x:z$
  - The operation  $@Y$  returns the variable  $w$   
(assuming the environment  $\{X \rightarrow x, Y \rightarrow y, Z \rightarrow z, W \rightarrow w\}$ )

15

## Imperative programming



- By adding cells, we have **left functional programming** and **entered imperative programming**
  - Imperative paradigm = functional paradigm + cells
- Imperative programming allows programs to express and observe growth and change
  - This gives new ways of thinking that were not possible in functional programming
- Imperative programming is important for object-oriented programming (OOP)
  - OOP has new ways of structuring programs that are essential for building large systems

16



# Kernel language of imperative programming



- $\langle s \rangle ::= \text{skip}$   
 $\quad | \langle s \rangle_1 \langle s \rangle_2$   
 $\quad | \text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end}$   
 $\quad | \langle x \rangle_1 = \langle x \rangle_2$   
 $\quad | \langle x \rangle = \langle v \rangle$   
 $\quad | \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\quad | \{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$   
 $\quad | \text{case } \langle x \rangle \text{ of } \langle p \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\quad | \{\text{NewCell } \langle y \rangle \langle x \rangle\}$   
 $\quad | \langle x \rangle := \langle y \rangle$   
 $\quad | \langle y \rangle = @ \langle x \rangle$
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{procedure} \rangle \mid \langle \text{record} \rangle$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{procedure} \rangle ::= \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}$
- $\langle \text{record} \rangle, \langle p \rangle ::= \langle \text{lit} \rangle \mid \langle \text{lit} \rangle (\langle f \rangle_1 : \langle x \rangle_1 \dots \langle f \rangle_n : \langle x \rangle_n)$

17

# Kernel language of imperative programming



- $\langle s \rangle ::= \text{skip}$   
 $\quad | \langle s \rangle_1 \langle s \rangle_2$   
 $\quad | \text{local } \langle x \rangle \text{ in } \langle s \rangle \text{ end}$   
 $\quad | \langle x \rangle_1 = \langle x \rangle_2$   
 $\quad | \langle x \rangle = \langle v \rangle$   
 $\quad | \text{if } \langle x \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\quad | \{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$   
 $\quad | \text{case } \langle x \rangle \text{ of } \langle p \rangle \text{ then } \langle s \rangle_1 \text{ else } \langle s \rangle_2 \text{ end}$   
 $\quad | \{\text{NewCell } \langle y \rangle \langle x \rangle\}$   
 $\quad | \{\text{Exchange } \langle x \rangle \langle y \rangle \langle z \rangle\}$
- $\langle v \rangle ::= \langle \text{number} \rangle \mid \langle \text{procedure} \rangle \mid \langle \text{record} \rangle$
- $\langle \text{number} \rangle ::= \langle \text{int} \rangle \mid \langle \text{float} \rangle$
- $\langle \text{procedure} \rangle ::= \text{proc } \{ \$ \langle x \rangle_1 \dots \langle x \rangle_n \} \langle s \rangle \text{ end}$
- $\langle \text{record} \rangle, \langle p \rangle ::= \langle \text{lit} \rangle \mid \langle \text{lit} \rangle (\langle f \rangle_1 : \langle x \rangle_1 \dots \langle f \rangle_n : \langle x \rangle_n)$

## Second version

Both versions are equally expressive (since Exchange can be expressed with @ and := and vice versa), but the second version is more convenient for concurrent programming

$\langle y \rangle = @ \langle x \rangle$  and  $\langle x \rangle := \langle z \rangle$   
(atomically, i.e., as one operation)

18

# Mutable state is needed for modularity



19

## Mutable state is needed for modularity



- Before looking at data abstraction and object-oriented programming, let's take a closer look at what mutable state is good for
- We say that a program (or system) is **modular** with respect to a given part if that part can be changed without changing the rest of the program
  - “part” = function, procedure, component, module, class, library, package, file, ...
- We will show by means of an example that the use of mutable state allows us to make a program modular
  - This is not possible in the functional paradigm

20

## A scenario (1)



- Once upon a time there were three developers, P, U1, and U2
- P has developed module M that implements two functions F and G
- U1 and U2 are both happy users of module M

```
fun {MF} % Module definition
  fun {F ...}
    <Definition of F>
  end
  fun {G ...}
    <Definition of G>
  end
in 'export'(f:F g:G)
end
M = {MF} % Module instantiation
```

21

## A scenario (2)



- One day, developer U2 writes an application that runs slowly because it does too much computation
- U2 would like to extend M to count the number of times F is called by the application
- U2 asks P to make this extension, but to keep it modular so that no programs have to be changed to use it

```
fun {MF}
  fun {F ...}
    <Definition of F>
  end
  fun {G ...}
    <Definition of G>
  end
in 'export'(f:F g:G)
end
M = {MF}
```

22

## Oops!



- This is **impossible** in functional programming, because F does not remember what happened in previous calls: **it cannot count its calls**
  - The only solution is to change the interface of F by adding two arguments,  $F_{in}$  and  $F_{out}$ :  
**fun** {F ...  $F_{in}$   $F_{out}$ }  $F_{out}=F_{in}+1$  ... **end**
  - The rest of the program has to make sure that the  $F_{out}$  of each call to F is passed as  $F_{in}$  to the next call of F
- This means that M's interface has changed
- **All M's users**, even U1, have to change programs
  - U1 is especially unhappy, since it makes a lot of extra work for nothing

23

## Solution using a cell



- Create a cell when MF is called and increment it inside F
  - Because of static scope, the cell is hidden from the rest of the program: **it is only visible inside M**
- M's interface is extended without changing existing calls
  - M.f stays the same
  - A new function M.c appears that can safely be ignored
- P, U1, and U2 live happily ever after

```
fun {MF}  
  X = {NewCell 0}  
  fun {F ...}  
    X := @X+1  
    <Definition of F>  
  end  
  fun {G ...}  
    <Definition of G>  
  end  
  fun {Count} @X end  
in 'export'(f:F g:G c:Count)  
end  
M = {MF}
```

24

## Comparison



- **Functional programming:**
  - + A component never changes its behavior (**correctness is permanent**)
  - – Updating a component often means that its interface changes and therefore many other components must be updated
- **Imperative programming:**
  - + A component can be updated without changing its interface and so without changing the rest of the program (**modularity**)
  - – A component can change its behavior because of past calls (for example, it might break)
- Sometimes it is possible to combine both advantages
  - Use mutable state to manage updates, but make sure that the behavior of components does not change

25

## Data abstraction



26

## Data abstraction



- Data abstraction is the main organizing principle for building complex software systems
  - Without data abstraction, computing technology would stop dead in its tracks
- We will study what data abstraction is and how it is supported by the programming language
  - The first step toward data abstraction is called encapsulation
  - Data abstraction is supported by language concepts such as higher-order programming, static scoping, and explicit state

27

## Encapsulation



- The first step toward data abstraction, which is the basic organizing principle for large programs, is **encapsulation**
- Assume your television set is not enclosed in a box
  - All the interior circuitry is exposed to the outside
  - It's lighter and takes up less space, so it's good, right? NO!
- It's **dangerous for you**: if you touch the circuitry, you can get an electric shock
- It's **bad for the television set**: if you spill a cup of coffee inside it, you can provoke a short-circuit
  - If you like electronics, you may be tempted to tweak the insides, to "improve" the television's performance
- So it can be a good idea to put the television in an enclosing box
  - A box that protects the television against damage and that only authorizes proper interaction (on/off, channel selection, volume)

28

## Encapsulation in a program



- Assume your program uses a stack with the following implementation:

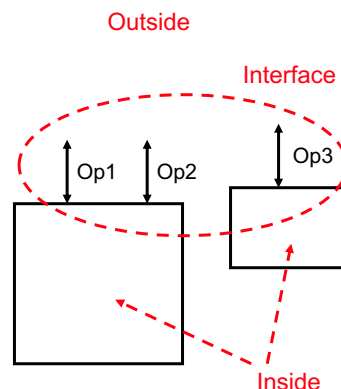
```
fun {NewStack} nil end
fun {Push S X} X|S end
fun {Pop S X} X=S.1 S.2 end
fun {IsEmpty S} S==nil end
```
- This implementation is not encapsulated!
  - It has the same problems as a television set without enclosure
  - It is implemented using lists that are not protected
    - A user can read stack values without the implementation knowing
    - A user can create stack values outside of the implementation
- There is no way to guarantee that an unencapsulated stack will work correctly
  - The stack must be encapsulated → data abstraction

29

## Definition of data abstraction



- A data abstraction is a part of a program that has an **inside**, an **outside**, and an **interface** in between
- The **inside** is hidden from the outside
  - All operations on the inside must pass through the interface, i.e., the data abstraction must use **encapsulation**
- The **interface** is a **set of operations that can be used according to certain rules**
  - Correct use of the rules guarantees that the results are correct
- The **encapsulation** must be **supported by the programming language**
  - We will see how the language can support encapsulation, that is, how it can enforce the separation between inside and outside



30

## Advantages of data abstraction



- A **guarantee** that the abstraction will work correctly
  - The interface only allows well-defined interaction with the inside
- A **reduction of complexity**
  - The user does not have to know the implementation, but only the interface, which is generally much simpler
  - A program can be partitioned into many independent abstractions, which greatly simplifies use
- The development of **large programs** becomes possible
  - Each abstraction has a **responsible developer**: the person who implements it, maintains it, and guarantees its behavior
  - Each responsible developer only has to **know the interfaces** of the abstractions used by the abstraction
  - It's possible for **teams of developers** to develop large programs

31

## The two main kinds of data abstraction



- There are two main kinds of data abstraction, namely **objects** and **abstract data types**
  - An object **groups together value and operations** in a single entity
  - An abstract data type **keeps values and operations separate**
- Some real world examples
  - **A television set is an object**: it can be used directly through its interface (on/off, channel selection, volume control)
  - **Coin-operated vending machines are abstract data types**: the coins and products are the values and the operations are the vending machines
- We will look at both objects and ADTs
  - Each has its own advantages and disadvantages

32



# Abstract data types



33

## Abstract data types



- An ADT consists of a set of values and a set of operations
- A common example: integers
  - Values: 1, 2, 3, ...
  - Operations: +, -, \*, div, ...
- In most of the popular uses of ADTs, the values and operations have no state
  - The values are **constants**
  - The operations have **no internal memory** (they don't remember anything in between calls)

34

## A stack ADT



- We can implement a stack as an ADT:
  - Values: all possible stacks and elements
  - Operations: NewStack, Push, Pop, IsEmpty
- The operations take (zero or more) stacks and elements as input and return (zero or more) stacks and elements as output
  - $S = \{\text{NewStack}\}$
  - $S2 = \{\text{Push } S \ X\}$
  - $S2 = \{\text{Pop } S \ X\}$
  - $\{\text{IsEmpty } S\}$
- For example:
  - $S = \{\text{Push } \{\text{Push } \{\text{NewStack}\} \ a\} \ b\}$  returns the stack  $S = [b \ a]$
  - $S2 = \{\text{Pop } S \ X\}$  returns the stack  $S2 = [a]$  and the top  $X = b$

35

## Unencapsulated implementation



- The stack we saw before is **almost** an ADT:
  - `fun {NewStack} nil end`
  - `fun {Push S X} X|S end`
  - `fun {Pop S X} X=S.1 S.2 end`
  - `fun {IsEmpty S} S==nil end`
- Here the stack is represented by a list
- But this is **not a data abstraction**, since the list is **not protected**
- How can we protect the list, and make this a true ADT?
  - How can we build an abstract data type with encapsulation?
  - We need a way to protect values

36

## Encapsulation using a secure wrapper



- To protect the values, we will use a **secure wrapper**:
  - The two functions Wrap and Unwrap will “wrap” and “unwrap” a value
  - $W = \{\text{Wrap } X\}$       % Given X, returns a protected version W
  - $X = \{\text{Unwrap } W\}$       % Given W, returns the original value X
- The simplest way to understand this is to consider that Wrap and Unwrap do **encryption and decryption using a shared key** that is only known by them
- We need a new Wrap/Unwrap pair for each ADT that we want to protect, so we use a procedure that creates them:
  - $\{\text{NewWrapper Wrap Unwrap}\}$  creates the functions Wrap and Unwrap
  - Each call to NewWrapper creates a pair with a **new shared key**
- We will not explain here how to implement NewWrapper, but if you are curious you can look in the book (Section 3.7.5)

37

## Implementing the stack ADT



- Now we can implement a true stack ADT:

```
local Wrap Unwrap in
  {NewWrapper Wrap Unwrap}

  fun {NewStack} {Wrap nil} end
  fun {Push W X} {Wrap X}{Unwrap W} end
  fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
  fun {IsEmpty W} {Unwrap W}==nil end
end
```
- How does this work? Look at the Push function: it first calls {Unwrap W}, which returns a stack value S, then it builds X|S, and finally it calls {Wrap X|S} to return a protected result
- Wrap and Unwrap are hidden from the rest of the program (static scoping)

38

## Final remarks on ADTs



- ADT languages have a long history
  - The language **CLU**, developed by Barbara Liskov and her students in 1974, is the first
  - This is only a little bit later than the first object-oriented language **Simula 67** in 1967
  - Both CLU and Simula 67 strongly influenced later object-oriented languages up to the present day
- ADT languages support a protection concept similar to Wrap/Unwrap
  - CLU has syntactic support that makes the creation of ADTs very easy
- Many object-oriented languages also support ADTs
  - For example, Java supports ADTs: Java integers are ADTs, and Java objects have some ADT properties

39

## Objects



40

## Objects



- A single object represents both a value and a set of operations
- **Example interface** of a stack object:

```
S={NewStack}  
{S push(X)}  
{S pop(X)}  
{S isEmpty(B)}
```

- The stack value is stored **inside** the object S
- **Example use** of a stack object:

```
S={NewStack}  
{S push(a)}  
{S push(b)}  
local X in {S pop(X)} {Browse X} end
```

41

## Implementing the stack object



- Implementation of the stack object:

```
fun {NewStack}  
  C={NewCell nil}  
  proc {Push X} C:=X|@C end  
  proc {Pop X} S=@C in C:=S.2 X=S.1 end  
  proc {IsEmpty B} B=(@C==nil) end  
in  
  proc {$ M}  
    case M of push(X) then {Push X}  
    [] pop(X) then {Pop X}  
    [] isEmpty(B) then {IsEmpty B} end  
  end  
end
```

- Each call to NewStack creates a **new stack object**
- The object is represented by a **one-argument procedure** that does **procedure dispatching**: a case statement chooses the operation to execute
- Encapsulation is enforced by **hiding the cell with static scoping**

42

## Stack as ADT and stack as object



- Here is the stack as ADT:

```
local Wrap Unwrap in
  {NewWrapper Wrap Unwrap}
  fun {NewStack} {Wrap nil} end
  fun {Push W X} {Wrap X}{Unwrap W} end
  fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
  fun {IsEmpty W} {Unwrap W}==nil end
end
```

- Here is the stack as object: (represented by a record)

```
fun {NewStack}
  C={NewCell nil}
  proc {Push X} C:=X|@C end
  proc {Pop X} S=@C in X=S.1 C:=S.2 end
  fun {IsEmpty} @C==nil end
in
  stack(push:Push pop:Pop isEmpty:IsEmpty)
end
```

- Any data abstraction can be implemented as an ADT or as an object

43

## Final remarks on objects



- Objects are omnipresent in computing today
- The first major object-oriented language was [Simula-67](#), introduced in 1967
  - It directly influenced [Smalltalk](#) (starting in 1971) and [C++](#) (starting in 1979), and through them, most modern object-oriented languages (Java, C#, Python, Ruby, and so forth)
- Most modern OO languages are in fact **data abstraction languages**: they incorporate both objects and ADTs
  - And other data abstraction concepts as well, such as components and modules
  - The Java language has both ADTs (e.g., Integer) and objects

44

# Four kinds of data abstraction



45

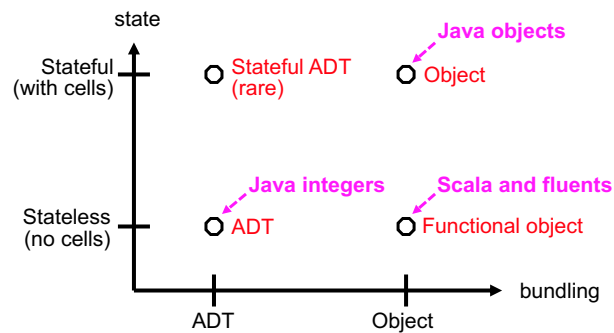
## Four kinds of data abstraction



- We have seen two kinds of data abstractions:
  - Abstract data types (without mutable state)
  - Objects (with mutable state)
- There are two other kinds of data abstractions
  - Abstract data types with state (stateful ADTs)
  - Objects without state (functional objects)
- This gives four kinds in all
  - Let's take a look at the two additional kinds
  - And then we'll conclude this lesson on data abstraction

46

## Four kinds of data abstraction



- Objects (with state) and ADTs (stateless) are in Java
- Functional objects are used in Scala and for big data
- Stateful ADTs are rarely used (so far!)

47

## The two less-used data abstractions



- A **functional object** is possible
  - Functional objects are immutable; invoking an object returns **another object with a new value**
  - Functional objects are becoming more popular
- A **stateful ADT** is possible
  - Stateful ADTs were much used in the C language (although without enforced encapsulation, since it is impossible in C)
  - They are also used in other languages (e.g., classes with static attributes in Java)
- Let's take a closer look at how to build them

48



## A functional object

- We can implement the stack as a functional object:

```
local
  fun {StackObject S}
    fun {Push E} {StackObject E|S} end
    fun {Pop S1}
      case S of X|T then S1={StackObject T} X end end
    fun {IsEmpty} S==nil end
  in
    stack(push:Push pop:Pop isEmpty:IsEmpty)
  end
in
  fun {NewStack} {StackObject nil} end
end
```

- This uses no cells and no secure wrappers. The simplest of all data abstractions since it **only needs higher-order programming**.



49

## Functional objects in Scala

- Scala is a hybrid functional-object language: it supports both the functional and object-oriented paradigms
- In Scala we can define an immutable object that returns another immutable object
  - For example, a RationalNumber class whose instances are rational numbers (and therefore immutable)
  - Adding two rational numbers returns another rational number
- Immutable objects are functional objects
  - The advantage is that they cannot be changed (the same advantage of any functional data structure)

50

## A stateful ADT



- Finally, let us implement our trusty stack as a stateful ADT:

```
local Wrap Unwrap
  {NewWrapper Wrap Unwrap}
  {NewStack} {Wrap {NewCell nil}} end
  fun {Push S E} C={Unwrap S} in C:=E|@C end
  fun {Pop S} C={Unwrap S} in
    case @C of X|S1 then C:=S1 X end
  end
  fun {IsEmpty S} @({Unwrap S})==nil end
in
  Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)
end
```

- This uses **both** a cell and a secure wrapper. Note that Push, Pop, and IsEmpty **do not need Wrap**! They modify the stack state by updating the cell *inside* the secure wrapper.

51

## Conclusion



- Data abstractions are a key concept needed for building large programs with confidence
  - Data abstractions are built on top of higher-order programming, static scoping, explicit state, records, and secret keys
  - Data abstractions are defined **precisely** in terms of these concepts; our definitions give the **semantics of data abstractions**
- There are **four kinds of data abstraction**, along two axes: **objects versus ADTs** on one axis and **stateful versus stateless** on the other
  - Two kinds are more visible than the others, but the others also have their uses (for example, functional objects are used in Scala)
- Modern programming languages strongly support data abstractions
  - They support much more than just objects; it is more correct to consider them **data abstraction languages** and not just object-oriented languages

52