Programming Language Concepts, CS2104 Lecture 10-11

Stateful Programming

Overview

- Stateful programming
  - what is state?
  - cells as abstract datatypes
  - the stateful model
  - relationship between the declarative model and the stateful model
  - indexed collections:
    - array model
  - parameter passing:
  - system building
    - component-based programming

Maintaining State

Encapsulated State I

Box O

An Interface that hides the state

State as a group of memory cells

Group of functions and procedures that operate on the state

- Box O can remember information between independent invocations, it has a memory
- Basic elements of <u>explicit</u> state
- Index datatypes
- Basic techniques and ideas of using state in program design

Encapsulated State II

Box O

An Interface that hides the state

State as a group of memory cells

Group of functions and procedures that operate on the state

- What is the difference between implicit state and explicit state?
- What is the difference between state in general and encapsulated state?
- Component based programming and object-oriented programming
- Abstract data types using encapsulated state

What is a State?

- State is a **sequence of values that evolves in time** that contains the intermediate results of a desired computation
- Declarative programs can also have state according this definition
- Consider the following program

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

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

What is an Implicit State?

The two arguments `Xs` and `A` represents an **implicit state**

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

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

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

What is an Explicit State?

An *explicit state* (in a procedure) is a state whose lifetime extends over more than one procedure call without being present in the procedure's arguments.

Extends beyond declarative programming model

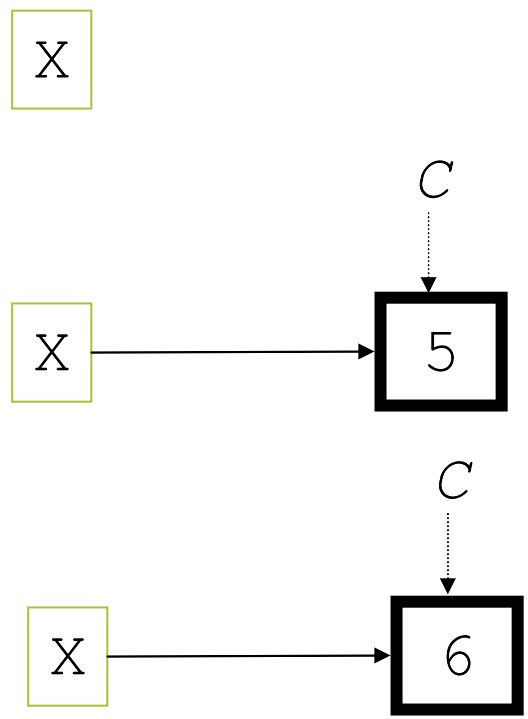
- support general concurrency
- support memory capability
- efficiency reasons

What is an Explicit State? Example

an unbound variable

A cell `C` is created with initial value `5`
`X` is bound to `C`
`@X` is bound to `5`

Cell `C` is assigned the value `6`,
`@X` is bound to `6`

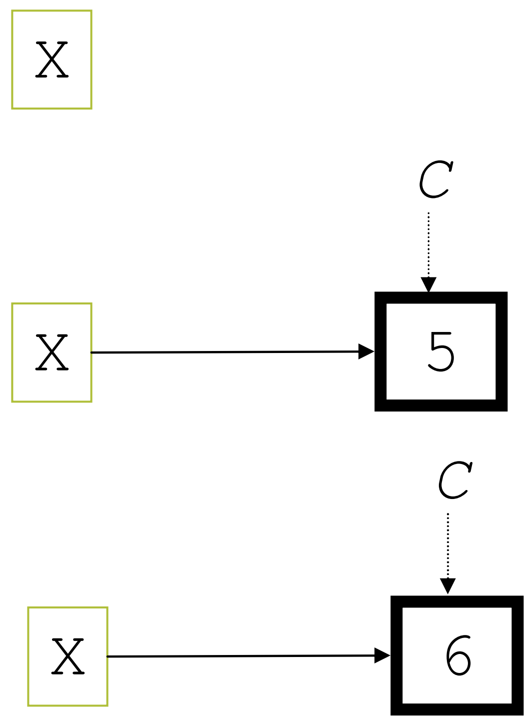


What is an Explicit State? Example

an unbound variable

A cell *C* is created with initial value `5`
`X` is bound to *C*
`@X` is bound to `5`

Cell *C* is assigned the value `6`,
`@X` is bound to `6`



- The cell is a value container with a unique **identity/address**
- `X` is really bound to the **identity/address** of the cell
- When the cell is assigned, `X` does not change

Maintaining State

- Agents maintain ***implicit*** state
  - state maintained as values passed as arguments
- Agents ***encapsulate*** state
  - state is only available within one agent
  - in particular, only one thread
- With ***cells*** we can have ***explicit*** state
  - programs can manipulate state by manipulating cells

Explicit State

- So far, the considered models do not have explicit state

- Explicit state is of course useful
    - algorithms might require state (such as arrays)
    - the right model for some task

Modular Approach to State

- Programs should be modular
  - composed from components
- Some components can use state
  - use only, if necessary
- Components from outside (interface) can still behave like functions

State: Abstract Datatypes

- Many useful abstractions are abstract datatypes using encapsulated state
  - arrays
  - dictionaries
  - queues
  - …

Cells

Cells as Abstract Datatypes

- `C={NewCell X}`
  - creates new cell `C`
  - with initial value `X`
- `X={Access C}` or equivalently `X=@C`
  - returns current value of `C`
- `{Assign C X}` or equivalently `C:=X`
  - assigns value of `C` to be `X`
- `{Exchange C X Y}` or equivalently `X=C:=Y`
  - atomically assigns `Y` into `C` and bind old value to `X`

Cells

- Are a model for explicit state
- Useful in few cases on itself
- Device to explain other stateful datatypes such as arrays

Examples

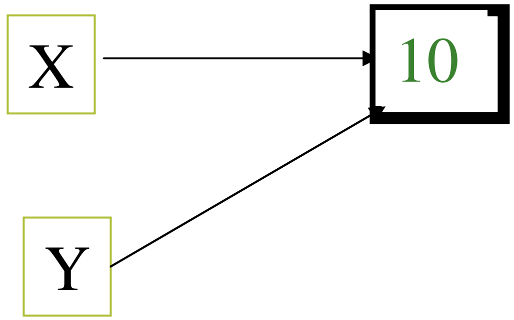
```
X = {NewCell 0}

{Assign X 5}

Y = X

{Assign Y 10}

{Access X} == 10 → true
X == Y  → true
```



Examples

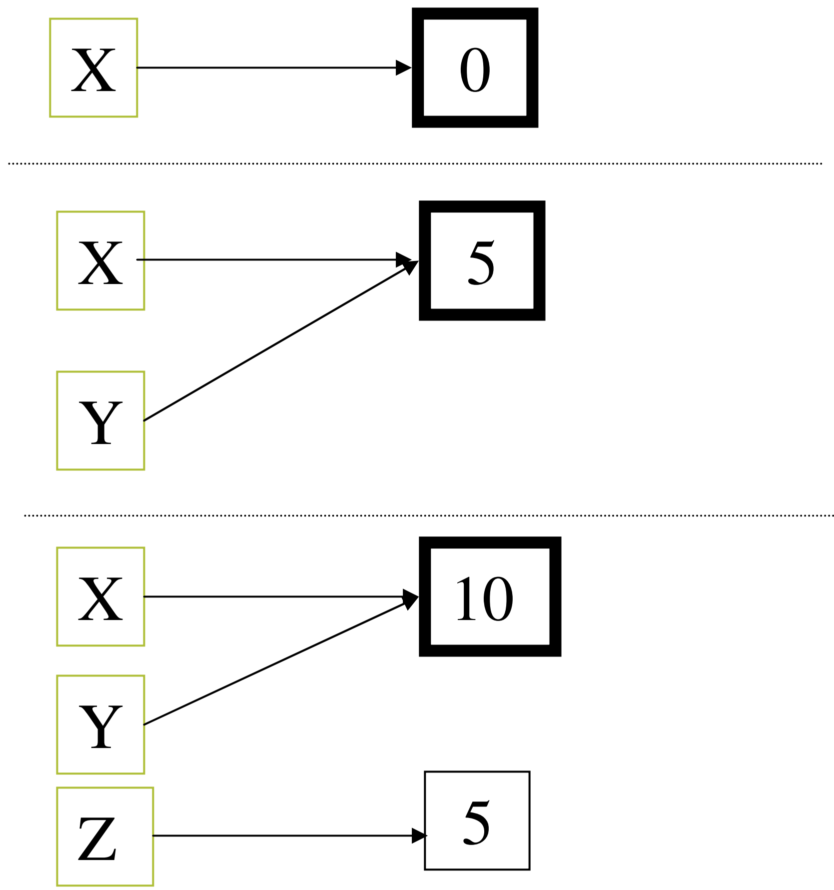
```
X = {NewCell 0}
```

```
{Assign X 5}
```

```
Y = X
```

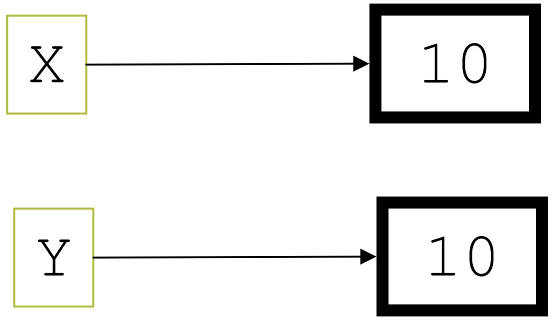
```
{Exchange Y Z 10}
```

`Z` is the old value of cell `Y`

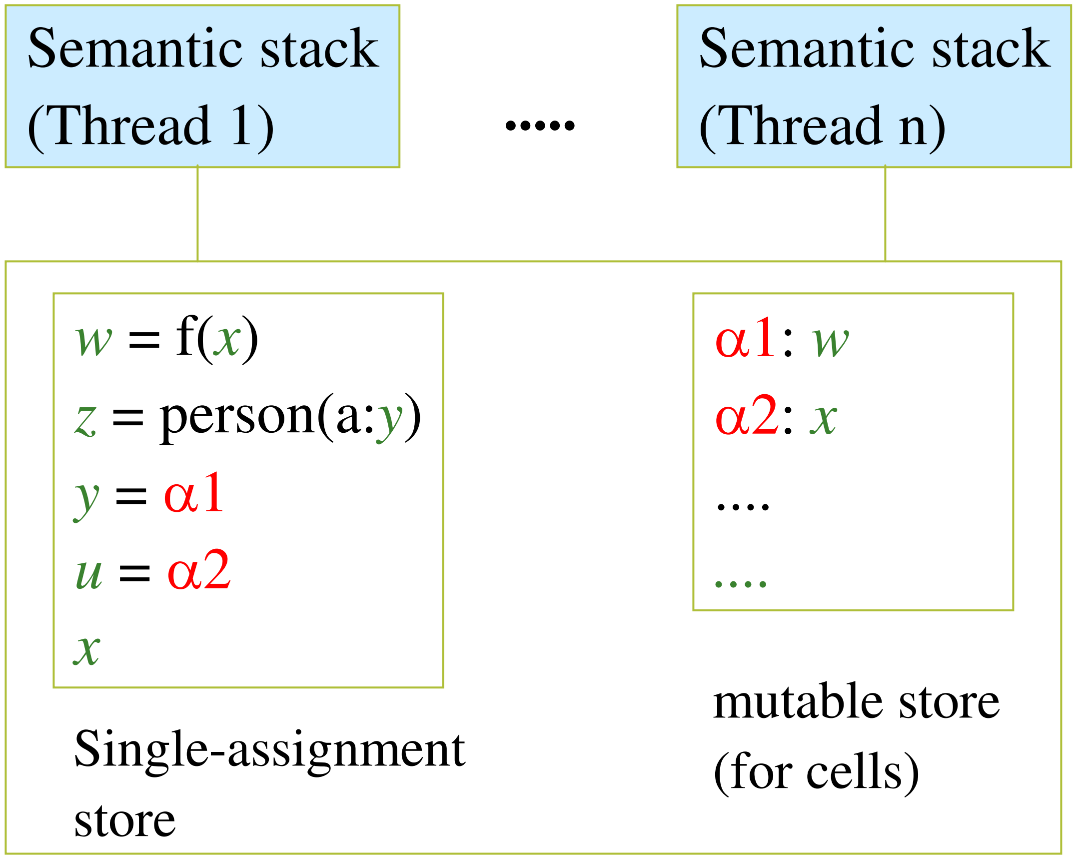


Examples

- `X = {NewCell 10}`
  `Y = {NewCell 10}`
- `X == Y % returns false`
- Because `X` and `Y` refer to different cells, with different identities
- `{Access X} == {Access Y}` returns `true`



Semantic Model Extended with Cells



The Stateful Model

$\langle \mathbf{s} \rangle$ ::=	**skip**	*empty statement*
|	$\langle \mathbf{s}_1 \rangle\ \langle \mathbf{s}_2 \rangle$	*statement sequence*
|	...	
|	**thread** $\langle \mathbf{s}_1 \rangle$ **end**	*thread creation*
|	`{NewCell` $\langle \mathbf{x} \rangle\ \langle \mathbf{c} \rangle$`}`	*cell creation*
|	`{Exchange` $\langle \mathbf{c} \rangle\ \langle \mathbf{x} \rangle\ \langle \mathbf{y} \rangle$`}`	*cell exchange*

The stateful model

| `{NewCell` ⟨**x**⟩ ⟨**c**⟩`}` *cell creation*
| `{Exchange` ⟨**c**⟩ ⟨**x**⟩ ⟨**y**⟩`}` *cell exchange*

`NewCell`: Create a new cell ⟨**c**⟩ with initial content ⟨**x**⟩

`Exchange`: Unify (bind) ⟨**x**⟩ to the old value of ⟨**c**⟩ and set the content of the cell ⟨**c**⟩ to ⟨**y**⟩

```
proc {Assign C X} {Exchange C _ X} end
fun {Access C} local X in
              {Exchange C X X} X end end
```

Do We Need Explicit State?

- Up to now the computation model we introduced in the previous lectures did not have any notion of explicit state
- An important question is: do we need explicit state?
- There are a number of reasons for introducing state, we discuss some of them here

Modular Programs

- A system (program) is **modular** if changes (updates) in the program are confined to the components where the functionality are changed
- Here is an example where introduction of explicit state in a systematic way leads to program modularity compared to programs that are written using only the declarative model (where every component is a function)

Encapsulated State I

- Assume we have three persons: `P`, `U1` and `U2`
- `P` is a programmer that developed a component `M` that provides two functions `F` and `G`
- `U1` and `U2` are system builders that use the component `M`

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

Encapsulated State II

- Assume we have three persons: `P`, `U1` and `U2`
- `P` is a programmer that developed a component `M` that provides two functions `F` and `G`
- `U1` and `U2` are system builders that use the component `M`

```
functor MF
export f:F g:G
define
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
end
```

Encapsulated State III

- User `U2` has a demanding application
- He wants to extend the module `M` to enable him to monitor how many times the function `F` is invoked in his application
- He goes to `P`, and asks him to do so without changing the interface to `M`

```
fun {M}
   fun {F ...}
        ⟨Definition of F⟩
   end
   fun {G ...}
        ⟨Definition of G⟩
   end
in 'export' (f:F g:G)
end
```

Encapsulated State IV

- This cannot be done in the declarative model, because `F` cannot remember its previous invocations
- The only way to do it there is to change the interface to `F` by adding two extra arguments `FIn` and `FOut`

```
fun {F ... +FIn ?FOut} FOut = FIn+1 ... end
```

- The rest of the program always remembers the previous number of invocations (`FIn` and `FOut`) returns the new number of invocation
- But this **changes** the interface!

Encapsulated State V

- A cell is created when `MF` is called
- Due to lexical scoping the cell is only visible to the created version of `F` and `Count`
- The `M.f` did not change
- New function `M.c` is available
- **`X` is hidden only visible inside `M` (encapsulated state)**

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

Relationship between the Declarative Model and the Stateful Model

- **Declarative programming** guarantees by construction that each procedure computes a function
- This means each component (and subcomponent) is a function
- It is possible to use encapsulated state (cells) so that a component is declarative from outside, and stateful from the inside
- Considered as a black-box the program procedure is still a function

Declarative versus Stateful

- Declarative:

```
declare X
thread X=1 end
thread X=2 end
{Browse X}
```

→ `1` or `2`, followed by a "`failure unification`"

- Stateful

```
declare X={NewCell 0}
thread X:=1 end
thread X:=2 end
{Browse @X}
```

→ `0`, `1`, or `2` depending on the order of threads execution

Programs with Accumulators

```
local
 fun {Sum1 Xs A}
  case Xs of X|Xr
    then {Sum1 Xr A+X}
   [] nil then A
  end
 end
in
 fun {Sum Xs}
  {Sum1 Xs 0}
 end
```

Programs with Accumulators

```
fun {Sum Xs}
 fun {Sum1 Xs A}
  case Xs of X|Xr
    then {Sum1 Xr A+X}
   [] nil then A
  end
 end
in
 {Sum1 Xs 0}
end
```

```
fun {Sum Xs}
 fun {Sum1 Xs}
  case Xs of X|Xr
     then
{Assign A X+{Access A}}
       {Sum1 Xr}
   [] nil then {Access A}
  end
 end
 A = {NewCell 0}
in
 {Sum1 Xs}
end
```

Programs with Accumulators

```
fun {Sum Xs}
 fun {Sum1 Xs}
  case Xs of X|Xr then
{Assign A X+{Access A}}
    {Sum1 Xr}
  [] nil then
    {Access A}
  end
 end
 A = {NewCell 0}
in
 {Sum1 Xs}
end
```

```
fun {Sum Xs}
  A = {NewCell 0}
in
 {ForAll Xs
    proc {$ X}
      {Assign A
        X+{Access A}}
    end}
  {Access A}
end
```

Another Declarative Function with State

```
fun {Reverse Xs}
  Rs={NewCell nil}
in
 for X in Xs do
        Rs:=X|@Rs end
 @Rs
end
```

`Rs` is a hidden internal state that do not live beyond the lifetime of above method.

Indexed Collections

- Indexed collections groups a set of (partial) values
- The individual elements are accessible through an index
- The declarative model provides:
  - tuples, e.g. date(17 december 2001)
  - records, e.g. date(day:17 month:december year:2001)
- We can now add state to the fields
  - arrays
  - dictionaries

Arrays

- An array is a **mapping** from integers to (partial) values
- The **domain** is a set of consecutive integers, with a *lower bound* and an *upper bound*
- The range can be mutated (change)
- A good approximation is to think of arrays as a tuple of cells

Array Model

- Simple array
  - fields indexed from `1` to *n*
  - values can be accessed, assigned, and exchanged
- Model: tuple of cells

Arrays

- `A={NewArray L H I}`
  - create array with fields from `L` to `H`
  - all fields initialized to value `I`
- `X={ArrayAccess A N}`
  - return value at position `N` in array `A`
- `{ArrayAssign A N X}`
  - set value at position `N` to `X` in array `A`
- `{ArrayExchange A N X Y}`
  - change value at position `N` in `A` from `X` to `Y`
- `A2={Array.clone A}`
  - returns a new array with same indices and contents as `A`

Example 1

- `A = {MakeArray L H F}`
- Creates an array `A` where for each index `I` is mapped to `{F I}`

```
fun {MakeArray L H F}
 A = {NewArray L H unit}
in
 for I in L..H do
   A.I := {F I}
 end
 A
end
```

Array2Record

- `R = {Array2Record L A}`
- Define a function that takes a label `L` and an array `A`, it returns a record `R` whose label is `L` and whose features are from the lower bound of `A` to the upper bound of `A`
- We need to know how to make a record
- `R = {Record.make L Fs}`
  - creates a record `R` with label `L` and a list of features (selector names), returns a record with distinct fresh variables as values
- `L = {Array.low A}` and `H = {Array.high A}`
  - Return lower bound and higher bound of array `A`

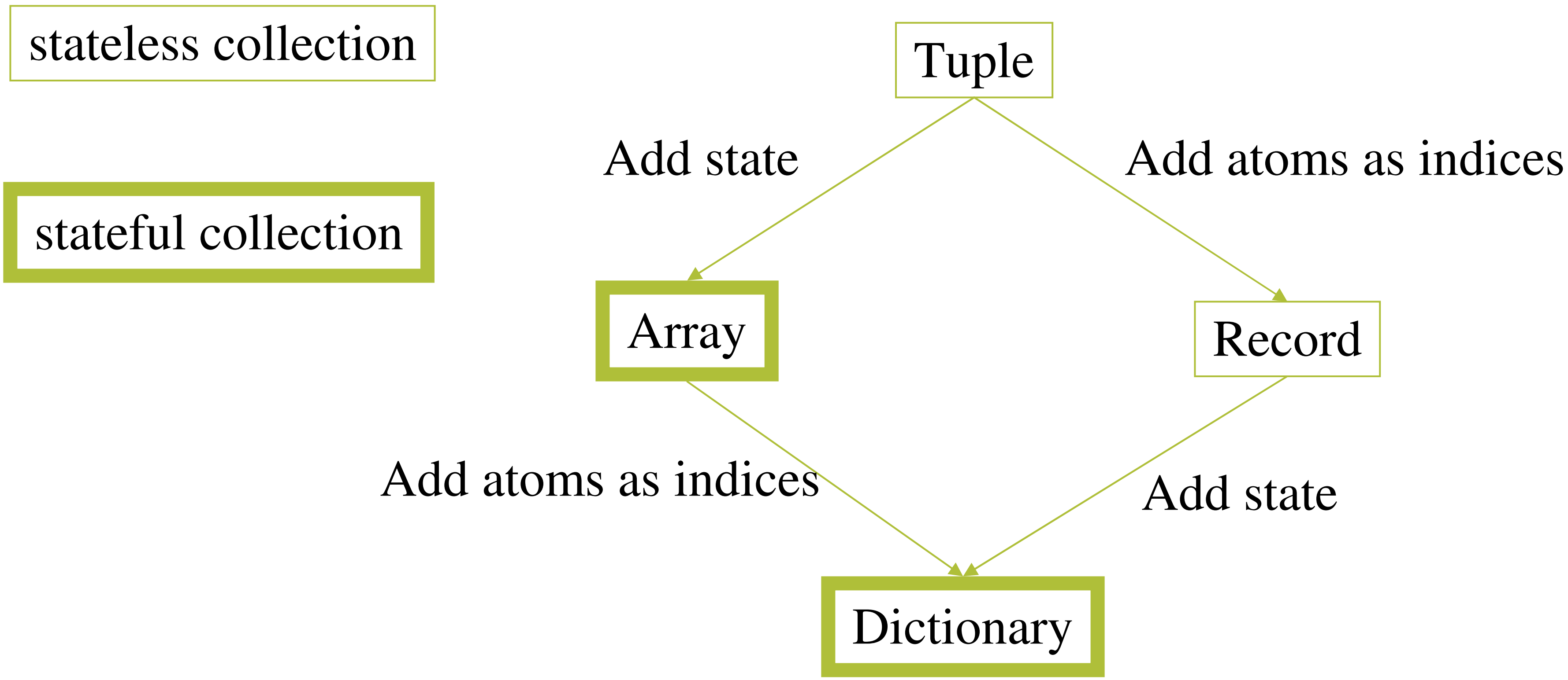
Array2Record. Example

```
fun {Array2Record LA A}
   L = {Array.low A}
   H = {Array.high A}
   R = {Record.make LA {From L H}}
in
   for I in L..H do
         R.I = A.I
   end
   R
end
```

Tuple to Array. Example

```
fun {Tuple2Array T}
   H = {Width T}
in
   {MakeArray 1 H
      fun{$ I} T.I end}
end
```

Indexed Collections



Other Collections

lists

potentially infinite lists

streams

ports

generalizes streams to stateful mailboxes

stateless collection

stateful collection

stacks

queues

stateless

Parameter Passing

- Variety of parameter passing mechanisms can be simulated using cells, e.g.
  - Call by Reference
  - Call by Variable
  - Call by Value
  - Call by Value-Result
  - Call by Name
  - Call by Need

Call by Reference

- Pass language entity to methods

- What is a language entity?
  - single-assignment variable
  - cell
  - local variable (in C)
    - Is it address? &v
    - Is it its value? v

Call by Variable

- Identity of cell is passed
- (special case of call by reference)

```
proc {Sqr A}
    A:=@A+1
    A:=@A*@A
end
```

```
local C={NewCell 0} in
  C:=5
  {Sqr C}
  {Browse @C}
end
```

Call by Value

- A value is passed and put into a local cell.

```
proc {Sqr A}
     D={NewCell A}
in   D:=@D+1
     D:=@D*@D
end
```

```
local C={NewCell 0} in
  C:=5
  {Sqr @C}
  {Browse @C}
end
```

Call by Value-Result

- A value is passed into local cell on entry of method, and passed out on exit of method.

```
proc {Sqr A}
     D={NewCell @A}
in   D:=@D+1
     D:=@D*@D
     A:=@D
end
```

```
local C={NewCell 0} in
  C:=5
  {Sqr C}
  {Browse @C}
end
```

Call by Name

- A function for each argument that returns a cell on invocation.

```
proc {Sqr A}
     {A}:=@{A}+1
     {A}:=@{A}*@{A}
end
```

```
local C={NewCell 0} in
  C:=5
  {Sqr fun {$} C end}
  {Browse @C}
end
```

Call by Need

- The function is called once and used multiple times.

```
proc {Sqr A}
     D={A}
in   D:=@D+1
     D:=@D*@D
end
```

```
local C={NewCell 0} in
  C:=5
  {Sqr fun {$} C end}
  {Browse @C}
end
```

System Building

- Abstraction is the best tool to build complex system
- Complex systems are built by layers of abstractions
- Each layer have two parts:
  - Specification, and
  - Implementation
- Any layer uses the specification of the lower layer to implement its functionality

Properties Needed to Support the Principle of Abstraction

- Encapsulation
  - Hide internals from the interface
- Compositionality
  - Combine parts to make new parts
- Instantiation/invocation
  - Create new instances of parts

Component-Based Programming

- Supports
  - Encapsulation
  - Compositionality
  - Instantiation

Object-Oriented Programming

- Supports
  - Encapsulation
  - Compositionality
  - Instantiation
- Plus
  - Inheritance

Maintainability Issues

- Component design
  - Encapsulate design decisions
  - Avoid changing component interfaces
- System design
  - Reduce external dependency
  - Reduce levels of indirection
  - Predictable dependencies
  - Make decisions at right level
  - Document violations

Features of Data Abstraction

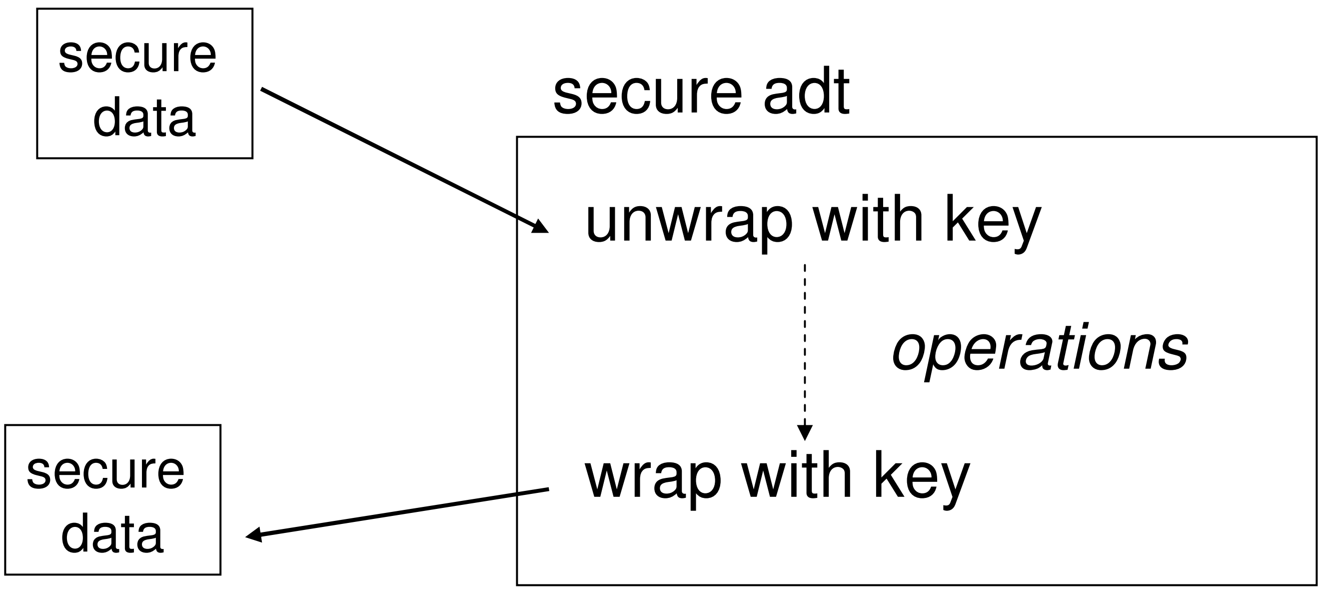
- Open/secure
  - Open – encapsulation enforced by programmer
  - Secure – implementation details not accessible to user
- Unbundled/bundled
  - Value/operations defined separately
  - Value/operation together, e.g. objects
- Explicit state/declarative
  - declarative – no mutable state

    e.g. `push :: {Stack A, A}` $\rightarrow$ `State A`

Making ADT Secure in Oz

- Make values secure using keys

`{NewName}`	return a fresh name
`N1==N2`	compares names `N1` and `N2`



Making ADT Secure in Oz

```
proc {NewWrapper ?Wrap ?Unwrap}
   Key={NewName}
in
  fun {Wrap X}
     fun {$ K} if K==Key then X
          else raise error end end
  end
  fun {Unwrap W}
     {W Key}
   end
end
```

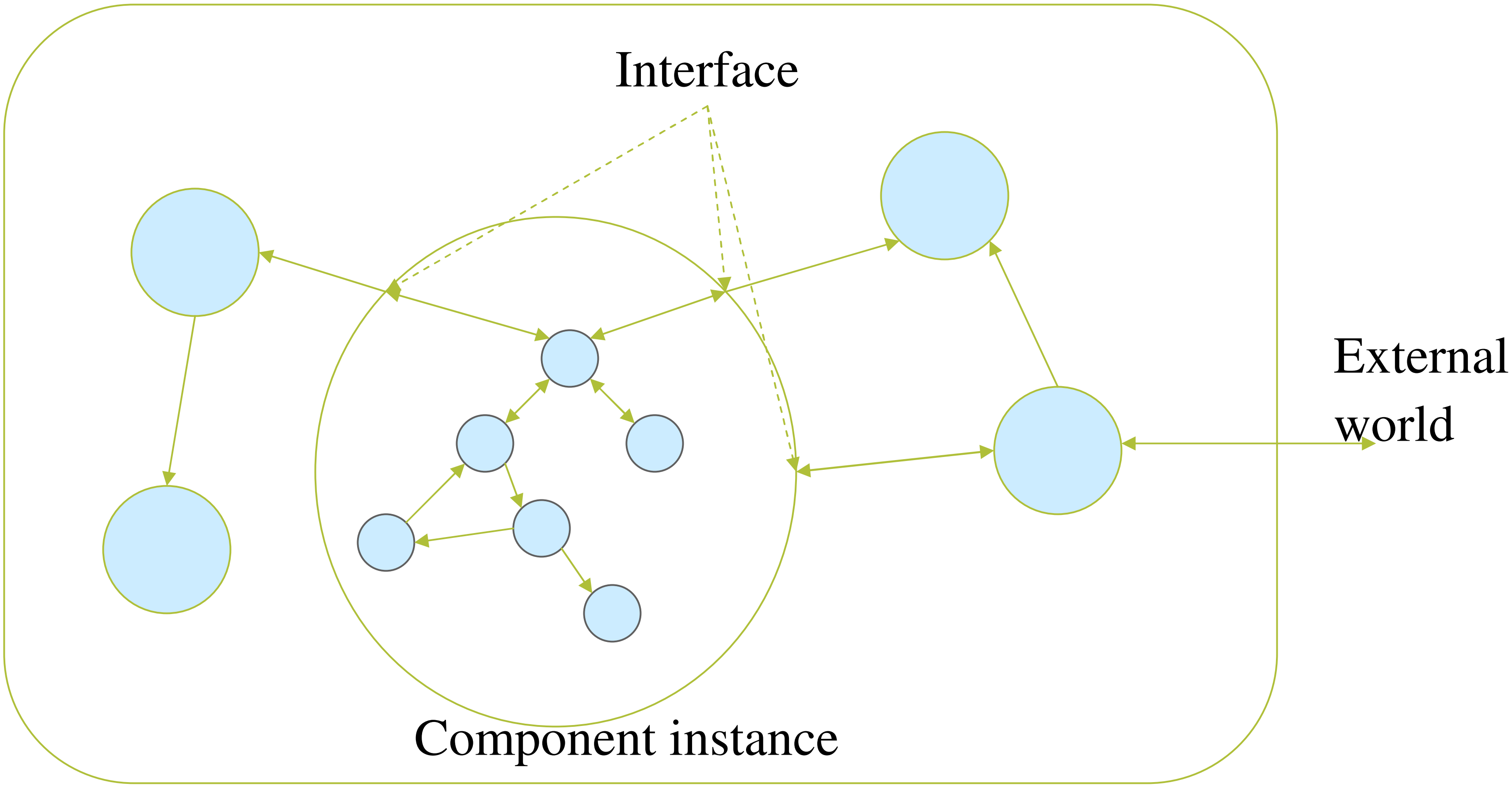
Using Security Wrapper in Stack ADT

```
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push S E} {Wrap E|{Unwrap S}} end
   fun {Pop S E}
       case {Unwrap S} of
            X|S1 then E=X {Wrap S1} end
   end
   fun {IsEmpty S} {Unwrap S}==nil end
end
```

Component-Based Programming

- ”Good software is good in the large and in the small, in its high level architecture and in its low-level details”. In Object-oriented software construction by Bernard Meyer
- What is the best way to build big applications?
- A large application is (almost) always built by a team
- How should the team members communicate?
- This depends on the application’s structure (architecture)
- One way is to structure the application as a hierarchical graph

Component-Based Programming



Component-Based Design

- Team members are assigned individual components
- Team members communicate at the interface
- A component, can be implemented as a record that has a name, and a list of other component instances it needs, and a higher-order procedure that returns a component instance with the component instances it needs
- A component instance has an interface and an internal entities that serves the interface

Model Independence Principle

- As the system evolves, a component implementation might change or even the model changes
  - declarative (functional)
  - stateful sequential
  - concurrent, or
  - relational
- The interface of a component should be independent of the computation model used to implement the component
- The interface should depend only on the externally visible functionality of the component

What Happens at the Interface?

- The power of the component based infrastructure depends to a large extent on the expressiveness of the interface
- How does components communicate with each others?
- We have three possible case:
  - The components are written in the same language
  - The components are written in different languages
  - The components are written in different computation model

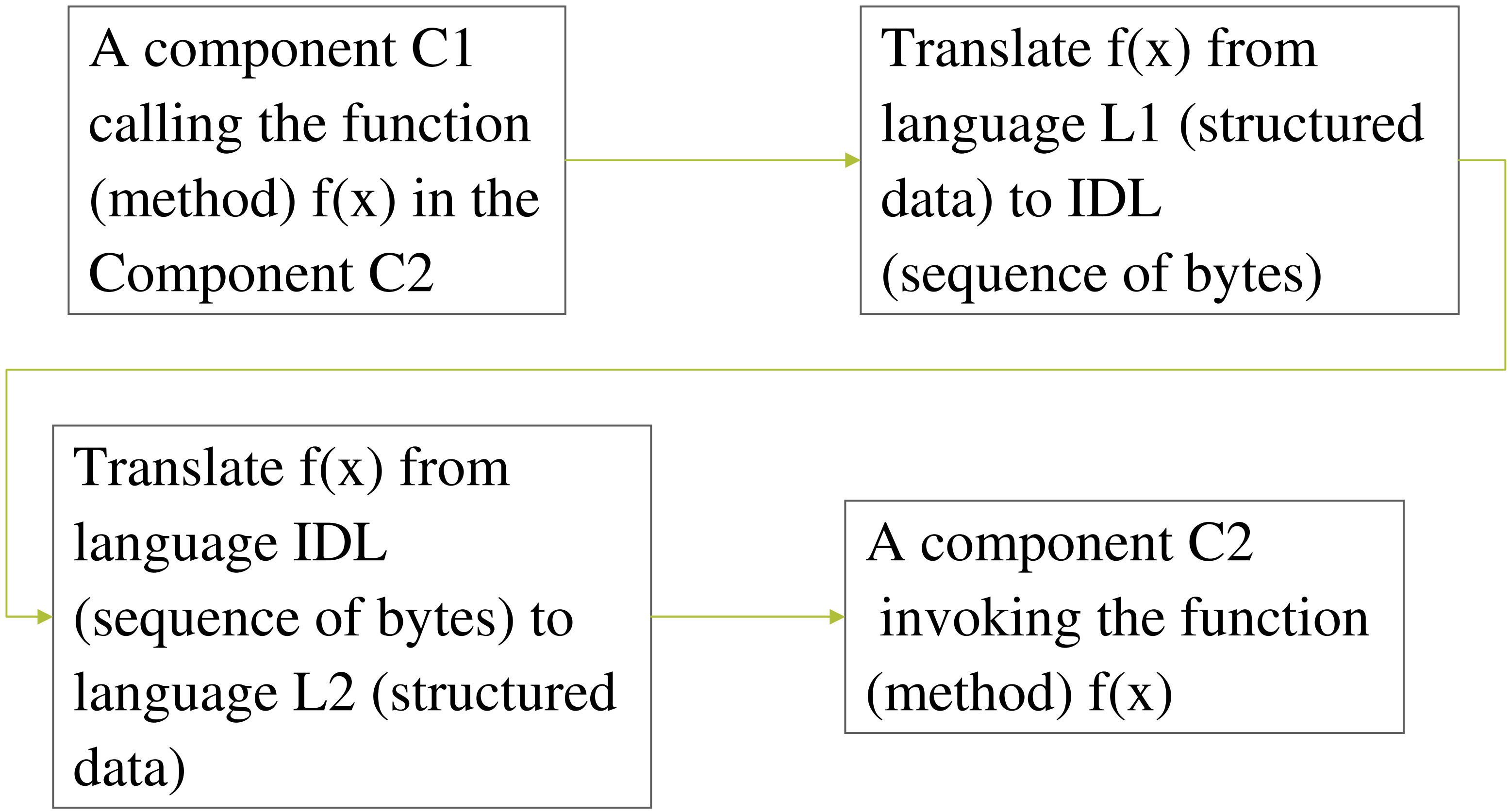
Components in the Same Language

- This is easy
- In Mozart/Oz, component instances are modules (records whose fields contain the various services provided by the component-instance part)
- In Java, interfaces are provided by objects (method invocations of objects)
- In Erlang, component instances are mainly concurrent processes (threads), communication is provided by sending asynchronous messages

Components in Different Languages

- An intermediate common language is defined to allow components to communicate given that the language provide the same computation model
- A common example is CORBA IDL (Interface Definition Language) which maps a language entity to a common format at the client component, and does the inverse mapping at the service-provider component
- The components are normally reside on different operating system processes (or even on different machines)
- This approach works if the components are relatively large and the interaction is relatively infrequent

Illustration (one way)



Summary

- Stateful programming
  - what is state?
  - cells as abstract datatypes
  - the stateful model
  - relationship between the declarative model and the stateful model
  - indexed collections:
    - array model
  - system building
    - component-based programming

Quiz 2

- 2^{nd} Nov 2007
- Duration : 1.5 hour
- All topics so far
- But focus on more on recent topics.
- Open-Book

Reading suggestions

- Chapter 6, Sections 6.1-6.3, 6.5, 6.7 from [van Roy,Haridi; 2004]
- Exercises 6.10.1-6.10.7 from [van Roy,Haridi; 2004]

Thank you for your attention!

Programs with Accumulators

```
fun {Sum Xs}
  A = {NewCell 0}
in
  {ForAll Xs
     proc {$ X}
       {Assign A
          X+{Access A}}
     end}
  {Access A}
end
```

```
fun {Sum Xs}
  A = {NewCell 0}
in
  for X in Xs do
   {Assign A
      X+{Access A}}
  end
  {Access A}
end
```

- The state is encapsulated inside each procedure invocation