Programming Language Concepts, cs2104 Lecture 12 (2003-11-07)

Seif Haridi

Department of Computer Science, NUS

haridi@comp.nus.edu.sg

Reading Suggestions

- Chapters 5
- And of course the handouts!

Exam Rules and Tips

- 120 minutes are 40 points!
- Read the entire exam first!
- Carefully study tutorials!

- Write your Matric. No. on each sheet!!!!!!

Overview

- Previous lecture
  - agents with state
  - cells
  - state encapsulation
  - objects and classes

- This lecture
  - repetition
  - overview
  - outlook
  - questions & answers

Agents With State

Agent Ingredients

- Agents
  - thread — independent computation
  - port — delivery address
  - stream — ordered mailbox
  - record — message, state
  - higher-order procedure — separate functionality from generic abstraction
- Also: ***active objects*** (object plus thread), servers

LCO Essential for Agents

- Last call-optimization is essential for agents…

…as they are supposed to run forever!

Why Do Agents Matter?

- Model to capture communicating entities
- Each agent is simply defined in terms of how it replies to messages
- Each agent has a thread of its own
  - no screwup with concurrency
  - we can possible have state for each agent easily!
- ***Extremely useful to model systems!***

Agents as State Transformers

- Receive messages
- To be useful they need to maintain state
  - changing over time
- Model: agent *a* is modeled by function

  state × message → state

Defining an Agent

- Agent
  - how it reacts to messages
  - how it transforms the state
  - from which initial state it starts
- Additionally
  - agent might compute by sending messages to other agents

Example: A Cell Agent

```
fun {CellProcess S M}
   case M
   of assign(New) then
      New
   [] access(Old) then
      Old=S S
   end
end
```


The `NewAgent` Abstraction

- To create a new agent with state
  - create a port for receiving messages
  - create a thread that executes agent
  - start agent by applying its function on an initial state
  - agent made available by the port

NewAgent

```
fun {NewAgent Process InitState}
   Port Stream
in
   Port={NewPort Stream}
   thread
     %% Execute agent…
   end
   Port
end
```


Executing the Agent

```
thread
   proc {Execute State Stream}
      case Stream of M|Rest then
         NewState={Process State M}
      in
         {Execute NewState Rest}
      end
   end
in
   {Execute InitState Stream}
end
```

Executing With FoldL

```
thread FinalState in
   FinalState ={FoldL Stream Process InitState}
end
```


Complete NewAgent

```
fun {NewAgent Process InitState}
   Port Stream
in
   Port={NewPort Stream}
   thread FinalState in
     FinalState ={FoldL Process Stream
                        InitState}
   end
   Port
end
```

Creating a Cell Agent

```
declare
CA = {NewAgent CellProcess 0}
```

- Cell agent
  - initialized with zero as state

State and Concurrency

- Difficult to guarantee that state is maintained consistently in a concurrent setting
- Typically needed: atomic execution of several statements together
- Agents guarantee atomic execution

Other Approaches

- Atomic exchange
  - Low-level
  - hardware: test-and-set
- Locks: allow only one thread in a “critical section”
  - monitor: allows multiple threads within a critical section, but only one active at a time
  - generalizations to single writer/multiple reader

Maintaining State

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 models considered do not have explicit state
- Explicit state is of course useful
  - algorithms might require state (such as arrays)
  - the right model for the task

Using 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: Example

- Lab assignment 3: computing the frequency map
  - consider all bytes, increase frequency by creating a new record with a just a single field changed
- Much more efficient: using state with dictionaries

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}`
  `X=@C` (alternative notation)
  - returns current value of `C`
- `{Assign C X}`
  `C := X` (alternative notation)
  - assigns value of `C` to be `X`
- `{Exchange C X Y}`
  `X = C := Y` (alternative notation)
  - atomically assigns `C` to `Y` and returns old value `X`

Cells

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

Array Model

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

Arrays

- `A={NewArray N I}`
  - create array with fields from 1 to `N`
  - all fields initialized to value `I`
- `X={ArrayAccess A N}`
  `X=A.N` (alternative notation)
  - return value at position `N` in array `A`
- `{ArrayAssign A N X}`
  `A.N:=X` (alternative notation)
  - set value at position `N` to `X` in array `A`
- `{ArrayExchange A N X Y}`
  `X=A.N:=Y`
  - exchange value at position `N` in `A` from `X` to `Y`

Homework

- Implement array abstract datatype
  - use tuple of cells

Ports Revisited

Ports

- Provide send operation
- Always maintain tail of stream
- Sending is appending cons with message

How to Program Ports?

- Idea: cell keeps current tail of stream
  - invariant: cell keep unassigned variable!
- Sending
  - access current tail of stream
  - appending message
  - assign current tail of stream

…must of course to be atomic with exchange!
what could happen otherwise?

Ports from Cells: Bad

```
fun {NewPort Stream}
   {NewCell Stream}
end
proc {Send P M}
   Old New
in
   Old = P := New % atomic exchange
   Old = M|New
end
```


Why Bad?

- Port is a cell in previous solution
- I can break the port abstraction…
- How: just put some junk into the cell

```
{Assign P crash}
```

Real Ports…

- Abstraction cannot be compromised
  - sending messages always works
- However: everybody is allowed to send messages to a port

How to Fix Ports from Cells

- Confine the cell to the send operation
  - by lexical scoping
- Invariant can never be broken

Ports from Cells

```
fun {Port Stream}
    C={NewCell Stream}
    proc {Send M}
       Old New
    in
       Old = C := New
       Old = M|New
    end
in
   port(send:Send)
end
```


Ports from Cells

```
fun {Port Stream}
    C={NewCell Stream}
    proc {Send M}
       Old New
    in
       Old = C := New
       New = M|Old
    end
in
   port(send:Send)
end
```

only Send
can use cell

Ports from Cells

- Returns the send procedure

- Example of use:

```
S={Port Xs}
{Browse Xs}
{S.send a}
{S.send b}
```


State Encapsulation

- State is encapsulated by construction of `Port`

- Encapsulation
  - guarantees invariant (cells maintains stream tail)
  - makes the abstraction “secure”

Objects and Classes

Objects

- Object is one convenient way of encapsulating state
  - only methods can access state
  - important invariants can be secured
- As well as very useful to model objects of the real world

Model for Objects

- Methods are procedures
  - have access to state
  - restrict access to state
- State of an object
  - record of cells
  - similar to our construction of arrays

Classes

- Describe how objects are constructed
  - initial state
  - methods
- Classes can be constructed from other classes by inheritance

Classes and Objects

- A full-blown object system can be obtained easily from
  - records — state field-name
  - cells — state field-value
  - procedures — methods
  - lexical scoping — access from methods to state
- First-class procedures are very powerful
  - allow to program inheritance and object creation

State: Abstract Datatypes

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

Course Summary

Content

- Study of programming models
  - declarative programming model
  - declarative concurrency
  - message sending and state
- Study of programming techniques
  - recursion
  - iterative computations
  - generic programs: higher-order programming
  - …
- Tools for analyzing and understanding programs
  - abstract machines
  - …
  - …

Declarative Programming Model

Most Important Concepts

- Single-assignment variables
  - partial values
- Abstract machine
  - a *tool* for understanding computations
  - a *model* of computation
  - based on environments
  - supports last call optimization
- Procedures with contextual environment
- Full versus kernel language

Abstract Machine

- General approach to explain how programming language computes
  - **model** for computation
- Can serve as base for implementation
  - pioneered by Prolog D.H.D. Warren, 1980's
  - many other languages including Oz
  - recent: JVM (Java) SUN
    CLR (C#, …) Microsoft

Declarative means…

- Programs returns
  same result
  for
  same arguments
- Always, always, always…
  regardless of any other computations

Declarative Programming Properties

- Independence
  - write programs independently
  - test and debug independently
  - other components of program do not matter
- Simple reasoning
  - declarative programs only compute values
  - no hidden state, no history, …
- This means simple development…

Be as Declarative as You Can

- Many program components can be written in a declarative style
  - use the benefits as much as possible
- For the rest, use other techniques
  - concurrency
  - state
  - objects

Significance

- Some languages are better than others at declarative programming (Oz versus C++)
- Declarative programming techniques are useful whatever language you program in
  - this course wants to sharpen your mind
  - this course uses a language that is good at declarative programming and the other techniques to come

Important Techniques and Concepts

Language Syntax

- Describe syntax of computer languages
  - lexical — words
  - syntactical — sentences
- Defined by grammars

Language Semantics

- How do programs compute?
- Model here: abstract machine
- Essential for:
  - understanding
  - transformations (accumulator, state invariants)
  - determining memory and runtime

Iterative Computations

- Iterative computations
  - computations that run with constant stack space
- Making computations iterative
  - using accumulators
  - understanding and design with state invariants
- Last call optimization
  - needed for iterative computations
  - special case: tail-recursion

Abstract Data Types

- Separate interface from implementation
- Sufficient to understand interface only
- Independence of implementation
  - as long as only interface is used
  - less knowledge required
  - independent development
  - software evolution

Generic Programs

- Make common program patterns generic
  - sorting
  - mapping
  - filtering
  - agent creation
  - …
- Use higher-order procedures
  - higher-order procedures are first-class citizens

Declarative Concurrency

The World Is Concurrent!

- Concurrent programs
  several activities execute
  simultaneously (concurrently)

- Most of the software you use is concurrent
  - operating system: IO, user interaction, many processes, …
  - web browser, Email client, Email server, …
  - telephony switches handling many calls
  - …

Thread-based Concurrency

- Threads
  - compute independently
  - share common abstract store
  - are lightweight
  - are scheduled fairly
  - have interleaving semantics

- Statements
  - automatically suspend and resume
  - computations triggered by dataflow variables

- Makes computations incremental

Semantics for Threads

- We insist on ***interleaving*** semantics
  - model: only one thread executes at a time
  - implementation: might execute several threads in parallel, however must execute as if one thread at a time
- Important property: ***monotonicity***
  - if a thread becomes runnable:

    …it stays runnable

    …doesn't matter when it is actually run

Demand-driven Execution

- Execute computations only if needed
  - infinite data structures
  - complex computations described easily
  - avoid computation as much as possible
- Expressed with "by-need" triggers
  - Computation executed ***at most*** once
  - Setup: threads + functions

Techniques and Concepts

- Producer, transducer, consumer
- Lazy functions
- Lazy streams
- Soft real-time programming

Agents and Message Sending

Agents and Message Sending

- Model problems as collection of multiple independent communicating entities
  - independent — concurrent, private thread
  - communicating — unique address, port
- Is not declarative: introduces
  - nondeterminism
  - state

Techniques and Concepts

- Agents with state
  - agents are state transformers
  - encapsulation of state
  - consistent management of state
- Protocols: rules for communication
  - involve multiple agents
  - simplicity is important (avoiding deadlocks)

Outlook

Questions & Answers

Which Thread Terminates First?

- Write a function
  - runs two computations
  - tells which one terminates first
- How?
  - passing computation
  - running computations
  - termination
  - finding out termination

Guarantee Execution

- Given: two statements
- How to guarantee that both are executed?

Asymptotic Complexity

- Valid statements only for particular computers?

Abstract Machine

- When is contextual environment computed?
  - procedure definition, or
  - procedure call
- Variable identifiers are always mapped to the same store variable
  - true or false?

Abstract Machine

- How is the environment constructed upon procedure call?
- What is environment adjunction/projection?
- Where is environment adjunction/projection used?

Higher-Order Programs

- Given: two unary functions F and G. Compute a unary function that executes the composition of F and G.

Accumulators

- Does asymptotic complexity always improve when making procedures tail-recursive?

Questions

- Activation condition for

  (**if** ⟨x⟩ **then** $\langle s\rangle_1$ **else** $\langle s\rangle_2$ **end**, *E*)

- Do all semantic statements share one environment?

- How many external references has a recursive procedure at least?

Questions

- Is there a guarantee that ⟨s⟩ in

  `thread` ⟨s⟩ **`end`**

  is actually executed?

- Is there a guarantee that $\langle s \rangle_2$ in

  `thread` $\langle s \rangle_1$ $\langle s \rangle_2$ **`end`**

  is actually executed?

Questions

- Is there a guarantee that $\langle s \rangle_2$ in

  `thread` $\langle s \rangle_1$ **`thread`** $\langle s \rangle_2$ **`end end`**

  is actually executed?

Questions

- What does function compute?

```
fun {Guess Xs Ys}
   case Xs
   of nil then Ys
   [] X|Xr then X|{Guess Ys Xr}
   end
end
```


Abstract Data Types

```
fun {NewBag}
   nil
end
fun {IsMember B X}
   …
end
fun {Add B X}
   X|B
end
```

Abstract Data Types: Use

```
fun {Extend B X}
   if {IsMember B X} then B
   else X|B
   end
end
```

- violates the abstract data type. Why?

Abstract Data Types: Correct Use

```
fun {Extend B X}
   if {IsMember B X} then B
   else {Add B X}
   end
end
```

End

The End

- Thank you for your kind attention
- It has been a pleasure for me!
- I hope…
  - …you learned something
  - …you enjoyed the course

Wishes

- Good luck with the exam

Have a Nice Weekend