Overview

- Previous lecture
  - message sending
  - protocols
  - soft real-time programming

- This lecture
  - agents with state
  - cells
  - ports revisited
  - state encapsulation
  - objects and classes

- Next lecture
  - overview
  - outlook
Agents and Message Passing Concurrency

Client-Server Architectures

- Server provides some service
  - receives message
  - replies to message
  - example: web server, mail server, ...

- Clients know address of server and use service by sending messages

- Server and client run independently
Peer-to-Peer Architectures

- Similar to Client-Server:
  - every client is also a server
  - communicate by sending messages to each other

- We call all these guys (client, server, peer) *agent*

- In the book this is called *portObject*

Common Features

- Agents
  - have identity
  - receive messages
  - process messages
  - reply to messages
  - mail address
  - mailbox
  - ordered mailbox
  - pre-addressed return letter

- Now how to cast into a programming language model?
Message Sending

- Message data structure
- Address port (oz concept)
- Mailbox stream of messages
- Reply dataflow variable in message

Port

- Port name:[S]
  - stores stream S under unique address (oz name)
  - stored stream changes over time

- The S is tail of message stream
  - sending a message M adds message to end
Message Sending to Port

- Port $a:[S]$
- Send $M$ to $a$
  - read stored stream $S$ from address $a$
  - create new store variable $S'$
  - bind $S$ to $M|S'$ (cons)
  - update stored stream to $S'$

- The state of the port changes over time
  - $a:[S]$ before sending the message $M$
  - $a:[S']$ after receiving the message $M$

Port Procedures

- Port creation
  $P=\{\text{NewPort } Xs\}$

- Message sending
  $\{\text{Send } P \ X\}$
Example

\begin{verbatim}
  declare S P
  P={NewPort S}
  {Browse S}

  • Displays initially S (or _)
\end{verbatim}

Example

\begin{verbatim}
  declare S P
  P={NewPort S}
  {Browse S}

  • Execute {Send P a}
  • Shows a _
\end{verbatim}
Example

declare S P
P={NewPort S}
{Browse S}

- Execute {Send P b}
- Shows a|b|_

Question

declare S P
P={NewPort S}
{Browse S}
thread {Send P a} end
thread {Send P b} end

- What will the Browser show?
Question

declare S P
P={NewPort S}
{Browse S}
thread {Send P a} end
thread {Send P b} end

● What will the Browser show?
● Either a|b|_ or b|a|_
  ● non-determinism: we can't say what

Answering Messages I

● Include the port P' of the sender in the message:

● \{Send P pair(Message P')\}

● Receiver sends answer message to P'

● \{send P' AnsMessage\}

● Traditional view
Answering Messages

- Do not reply by address, use something like pre-addressed reply envelope
  - dataflow variable!!!

- \{Send P pair(Message Answer)\}

- Receiver can bind Answer!
- Answer = AnsMessage

A Math Agent

proc {Math M}
    case M
      of add(N M A) then A = N + M
      [] mul(N M A) then A = N * M
      [] int(Formula A) then
        A = ...
      end
    end
end
Making the Agent Work

MP = {NewPort S}
proc {MathProcess Ms}
    case Ms of M|Mr then
        {Math M} {MathProcess Mr}
    end
end
thread {MathProcess S} end

Smells of Higher-Order...

proc {ForAll Xs P}
    case Xs
        of nil then skip
        [] X|Xr then {P X} {ForAll Xr P}
    end
end
• Call procedure P for all elements in Xs
Smells of Higher-Order...

- Using ForAll, we have

```plaintext
proc {MathProcess Ms}
  {ForAll Xs Math}
end
```

Making the Agent Work

```plaintext
MP = {NewPort S}
thread {ForAll S Math} end
```
Making the Agent Work

\[ MP = \{\text{NewPort S}\} \]
\[ \text{thread for M in S do } \{\text{Math M}\} \text{ end} \]
\[ \text{end} \]

Smells Even Stronger...

\[ \text{fun } \{\text{NewAgent Process}\} \]
\[ \quad \text{Port Stream} \]
\[ \quad \text{in} \]
\[ \quad \text{Port}=\{\text{NewPort Stream}\} \]
\[ \quad \text{thread } \{\text{ForAll Process}\} \text{ end} \]
\[ \quad \text{Port} \]
\[ \text{end} \]
Why Do Agents/Processes 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 screw-up with concurrency
  - we can easily extend the model so that each agent have a state (encapsulated)
- Extremely useful to model systems!

Summary

- Ports for message sending
  - use stream (list of messages) as mailbox
  - port serves as unique address
- Use agent abstraction
  - combines port with thread running agent
  - simple concurrency scheme
- Introduces non-determinism… and state!
Message Sending

Message Sending: Properties

- Message sending
  - asynchronous
  - ordered per thread
  - no order from multiple threads
  - first-class messages
Asynchronous Sending

\[ P = \{ \text{NewPort S} \} \]

thread ... \{Send P M\} ... end \hspace{1cm} (1)

thread ... \{Process S\} ... end \hspace{1cm} (2)

- Asynchronous: (1) continues immediately after sending
- Sender does not know when message processed
  - message processed eventually

Asynchronous Reply

- Sender sends message containing dataflow variable for answer
  - does not wait for receipt
  - does not wait for answer when sending

- Waiting for answer, only if answer needed

- Helps to avoid latency
  - sender continues computation
  - receiver might already deliver message
Synchronous Sending

- Sometimes more synchronization needed
  - sender wants to synchronize with receiver upon receipt of message
  - known as: *handshake, rendezvous*

- Can also be used for delivering reply
  - sender does not wait for reply computed, or
  - sender waits until reply computed

Waiting for Variables

- How to express that execution resumes only if variable $X$ bound?
- Notice that conditional is suspendable

```plaintext
proc {Wait X}
  if X==1 then skip else skip end
end
```
Synchronous Send

\begin{verbatim}
proc {SyncSend P M}
  Ack in {Send P M#Ack}
  {Wait Ack}
end

proc {Process MA}
case MA of M#Ack then
  Ack=okay ...
end
end
\end{verbatim}

Asynchronous Send

- Synchronous send can be turned into asynchronous send again by use of threads

\begin{verbatim}
proc {AsyncSyncSend P M}
  thread {SyncSend P M} end
end
\end{verbatim}

- Sending: variants can be mutually expressed
Message Order

- Order on same thread: A always before B
  
  ```
  thread
  ... {Send P A} ... {Send P B} ...
  end
  ```

- No order among threads
  
  ```
  thread ... {Send P A} ... end
  thread ... {Send P B} ... end
  ```

Messages

- Important aspect of agents
  - messages are first-class values: can be computed, tested, manipulated, stored
  - can contain any data structure including procedure values

- First-class messages are expressive
  - messages received stored in a log
  - agent forwards by adding time-stamp to message
A Compute Server

\[
\text{proc } \{\text{ComputeAgent M}\} \\
\text{case } M \\
\text{of } \text{run}(P) \text{ then } \{P\} \\
[] \text{ run}(F \ R) \text{ then } R=\{F\} \\
\text{end} \\
\text{end}
\]

- Runs as an agent in its own thread
- Executes procedures contained in messages

Distribution

- Spawn computations across several computers connected by network
- Message sending important way to structure distributed programs
- Compute servers make sense in this setting
- Oz: transparent distribution
Soft Realtime Programming

- Realtime
  - control computations by time
  - animations, simulations, timeouts, …

- Soft
  - suggested time
  - no time guarantees
  - no hard deadlines as for controllers, etc.
Delay

\{Delay N\}

suspends the thread for $N$ milliseconds

- Useful for building abstractions
  - timeouts
  - repeating actions
Protocols

- Protocol: rules for sending and receiving messages
  - programming with agents

- Examples
  - broadcasting messages to group of agents
  - choosing an agent

- Important properties of protocols
  - deadlock free

Broadcast

- Just send message M to all agents As

```plaintext
proc {Broadcast As M}
  {ForAll As proc {A}
    {Send A M}
    end}
end
```
Reminder: \textit{ForAll}

\begin{verbatim}
proc \{ForAll Xs P\}
  case Xs
    of nil then skip
    [X | Xr] then \{P X\} \{ForAll Xr P\}
  end
end
\end{verbatim}

Choosing an Agent

- Example: choosing the best lift
- More general: seeking agreement

- General idea:
  - Master Floor agent
    - send message to all slaves containing answer variable
  - Slaves Lift agents
    - answer by binding in the answer variable
    - if decision to be known, use dataflow variable again
Choosing an Agent

- **Master to one slave**
  
  \{Send S m(... Reply)\}
  
  ```plaintext
  case Reply of r(... Status) then
    if ... then Status=reserve
    else Status=reject
  end
  ```

- **Slave**
  
  ```plaintext
  case M of m(... Reply) then
    Reply=r(... Status)
    if Status==reject then ... else ... end
  ```

---

Choosing an Agent

- **Master:**
  
  - sends original message including variable for Reply
  - suspends until Reply bound

- **Slave:**
  
  - receives message
  - binds Reply, includes variable for Status
  - suspends until Status bound

- **Master:**
  
  - decides and binds Status

- **Slave:**
  
  - continues according to Status
Master to Multiple Slaves

Rs={Map Ss fun \$ S
  Reply in
  \{Send S m(... Reply)\}
  Reply
end}
Avoiding Deadlock

- Master can only proceed, after all slaves answered
  - will not process any more messages until then
- Slave can only proceed, after master answered
  - will not process any more messages until then

- What happens if multiple masters for same slaves?

Deadlock

Master
- M

Slave
- A
- B
- C
- D

Deadlock

Master  Slave

M  A
B  C
D  N

Deadlock

Master  Slave

M  A
B  C
D  N
Deadlock

Master

Slave

M

blocks!

N

blocks!

blocks!
**Deadlock**

- M blocks A, waits for B
- N blocks B, waits for A
- Deadlock!!!

**Avoiding Deadlock**

- Force all masters to send in order to all slaves:
  - First slave A, then B, then C, ...
  - Guarantee: If A available, all others will be available
  - That is as in lab assignment
- Use an adaptor
  - access to slaves through one single master
Summary

- Protocols for coordinating agents
- Can lead to deadlocks
- Simple structure best
- Details: Distributed Systems (and algorithms)
Next Lecture

- Agents with state
- State: cells and abstract data types
- Objects

Agents With State
Agents

- Receive messages

- To be useful they need to maintain state
  - changing over time

- Model: agent $a$ is modeled by function
  $$\text{state} \times \text{message} \rightarrow \text{state}$$

Agent States

```
<table>
<thead>
<tr>
<th>s0</th>
<th>a</th>
<th>s1</th>
<th>a</th>
<th>s2</th>
<th>a</th>
<th>...</th>
</tr>
</thead>
</table>
```

2003-10-31  S. Haridi, CS2104, L11 (slides: C. Schulte, S. Haridi)  64
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

```plaintext
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 an initial state
  - agent made available by the port

**NewAgent**

```plaintext
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

Structure of Execute

- Does `Execute` ring a bell with you:
  - starting from initial state
  - successively applying agent to state and message
  - agent is binary operation
  - ...
  -...
Structure of Execute

- Does Execute ring a bell with you:
  - starting from initial state
  - successively applying agent to state and message
  - agent is binary operation
  - ...

- Of course: we are folding!
  - which folding is it: left or right...

Reminder: FoldL

fun {FoldL Xs F S}
  case Xs
  of nil then S
  [] X|Xr then {FoldL Xr F {F S X}}
  end
end
Executing With FoldL

thread
  {FoldL Stream Process InitState}
end

● Wait!

Executing With FoldL

thread
  {FoldL Stream Process InitState}
end

● This is wrong! Why?
Executing With \texttt{FoldL}

\texttt{thread} \{\texttt{FoldL Stream Process InitState}\}
\texttt{end}

- Right: \texttt{FoldL} returns result…

Executing With \texttt{FoldL}

\texttt{thread} \texttt{Dummy in}
\texttt{Dummy=} \{\texttt{FoldL Stream Process InitState}\}
\texttt{end}
Complete `NewAgent`

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

Creating a Cell Agent

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

- Cell agent
  - initialized with zero as state
A Simple Task

proc \{Inc C\}
    N
in
    \{Send C access(N)\}
    \{Send C assign(N+1)\}
end

- Increment the cell’s content by one
  - Get the old value
  - Put the new value
- Does this work? NO! NO! Why?

A Simple Task Screwed...

C={NewAgent ...}
thread \{Inc C\} end
thread \{Inc C\} end

- We insist on result being 2!
  - sometimes: 2
  - sometimes: 1
- Why?
Execution Sketch A: Good

- Thread 1
  - executes access: value got: 0
- Thread 1
  - executes assign: value put: 1
- Thread 2
  - executes access: value got: 1
- Thread 2
  - executes assign: value put: 2

Execution Sketch B: Bad

- Thread 1
  - executes access: value got: 0
- Thread 2
  - executes access: value got: 0
- Thread 1
  - executes assign: value put: 1
- Thread 2
  - executes assign: value put: 1
What Is Needed

- We need to avoid that multiple access and assign operations get out of order
- We need to combine access and assign into one operation
  - we cannot guarantee that not interrupted
  - we can guarantee that state is correct
  - immediately put a dataflow variable
- Also: called atomic exchange
  - operation is atomic

A Cell Agent with Exchange

```latex
fun \{CellProcess S M\}
\begin{align*}
\text{case } M \text{ of}
\quad & \text{assign}(\text{New}) \text{ then } \text{New} \\
\quad & \text{access}(\text{Old}) \text{ then } \text{Old}=S \text{ Old} \\
\quad & \text{exchange}(\text{New Old}) \text{ then } \\
\quad & \quad \text{Old}=S \text{ New}
\end{align*}
\end{equation}
```
Incrementing Rectified

\textbf{proc} \{Inc \textit{C}\}
\begin{align*}
\text{New} & \quad \text{Old} \\
\text{in} & \\
\{ & \text{Send } \textit{C} \text{ exchange}(\text{New} \quad \text{Old})\} \\
\text{New} & = \quad \text{Old}+1 \\
\text{end} & \\
\end{align*}

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: use a lock together with a thread
  - 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

- \(C = \{\text{NewCell } X\}\)
  - creates new cell \(C\) with initial value \(X\)
- \(X = \{\text{Access } C\}\)
  - returns current value of \(C\)
- \(\{\text{Assign } C \ X\}\)
  - assigns value of \(C\) to be \(X\)
- \(\{\text{Exchange } C \ X \ Y\}\)
  - 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 datatypes 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

- \( A = \{\text{NewArray} \ N \ I\} \)
  - create array with fields from 1 to \( N \)
  - all fields initialized to value \( I \)

- \( X = \{\text{ArrayAccess} \ A \ N\} \)
  - return value at position \( N \) in array \( A \)

- \( \{\text{ArrayAssign} \ A \ N \ X\} \)
  - set value at position \( N \) to \( X \) in array \( A \)

- \( \{\text{ArrayExchange} \ A \ N \ X \ Y\} \)
  - 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 logic 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
   {Exchange P Old New}
   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 {NewSend Stream}
    C={NewCell Stream}
proc {Send M}
    Old New
    in
    {Exchange C Old New}
    Old = M|New
    end
in
    Send
end

only send can use cell
Ports from Cells

- Returns the send procedure

- Example of use:
  \[ S = \{ \text{NewSend} \ Xs \} \]
  \[ \{ \text{Browse} \ Xs \} \]
  \[ \{ S \ a \} \]
  \[ \{ S \ b \} \]

State Encapsulation

- State is encapsulated by construction of NewSend

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

Outlook

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

Have a Nice Weekend