Reading Suggestions

- Chapter 4
  - Sections 4.1-4.3 [careful]

- And of course the handouts!
Organizational

- Assignment 3 deadline is extended (again) to October 13 (Monday)
- Consultation at October 11 (Saturday) from 11:00 – 13:00 at PC Lab2
- Check your marks for Assignment 1
- Finish correction of midterm exam soon

Concurrency
Concurrency

- First: declarative concurrency
- What is concurrency?
- How to make a program concurrent?
- How do concurrent programs execute?

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
  - …
Why Should We Care?

- Software must be concurrent...
  ... for many application areas
- Concurrency can be helpful for constructing programs
  - organize programs into independent parts
  - concurrency allows to make them independent with respect to how they execute
  - essential: how do concurrent programs interact?
- Concurrent programs can run faster on parallel machines (including clusters)

Concurrent Programming Is Difficult...

- This is the traditional belief
- The truth is: concurrency is very difficult...
  ... if used with inappropriate tools and programming languages
- In particular troublesome: state and concurrency
  (see discussion end of Chapter 1)
Concurrent Programming Is Easy…

- Oz (as well as Erlang) has been designed to be very good at concurrency…

- Essential for concurrent programming here
  - data-flow variables
    - very simple interaction between concurrent programs, mostly automatic
  - light-weight threads

Declarative Concurrent Programming

- What stays the same
  - the result of your program
  - concurrency does not change the result

- What changes
  - programs can compute incrementally
  - incremental input… (such as reading from a network connection)… is processed incrementally
  - the fun: much greater!
Our Schedule

- Programming with threads
  - data-driven concurrency

- Demand driven execution
  - …so that you can do the next assignment

- Abstract machine
  - explain the details, very simple!
  - next lecture

Threads
Our First Program

```plaintext
declare X0 X1 X2 X3
thread X1=1 + X0 end
thread X3=X1 + X2 end
{Browse [X0 X1 X2 X3]}

- Browser will show [X0 X1 X2 X3]
  - variables are not yet assigned
```

- Both threads suspend
  - X1=1+X0 suspended
  - X3=X1+X2 suspended
  - X0 unassigned
  - X1, X2 unassigned
Our First Program

declare X0 X1 X2 X3

thread X1=1 + X0  end
thread X3=X1 + X2 end

{Browse [X0 X1 X2 X3]}

- Feeding X0=4

- Feeding X0=4
- first thread can execute, binds X1 to 5
Our First Program

declare X0 X1 X2 X3
thread X1=1 + X0 end
thread X3=X1 + X2 end
{Browse [X0 X1 X2 X3]}

- Feeding X0=4
  - first thread can execute, binds X1 to 5
  - Browser shows [1 5 X2 X3]

- Second thread is still suspending
  - variable X2 still not assigned
Our First Program

declare X0 X1 X2 X3
thread X1=1 + X0 end
thread X3=X1 + X2 end
{Browse [X0 X1 X2 X3]}

- Feeding X2=2
  - second thread can execute, binds X3 to 7
  - Browser shows [1 5 2 7]

Threads

- A thread is created by the thread ⟨s⟩ end statement

- Threads compute
  - independently
  - as soon as their statement can execute
  - interact by binding variables in store
The Browser

- The Browser is also implemented in Oz
- It runs in a thread of its own
- It also is run whenever browsed variables are bound
- It uses some extra functionality to look to unbound variables

The sequential model

Statements are executed sequentially from a single semantic stack

Single-assignment store

Semantic Stack

\[
\begin{align*}
  w &= a \\
  z &= \text{person(age: } y) \\
  x &= 42 \\
  y &= 42 \\
  u &= \ldots
\end{align*}
\]
The concurrent model

Multiple semantic stacks (threads)

Semantic Stack 1

Semantic Stack N

Single-assignment store

w = a
z = person(age: y)
x
y = 42
u

Concurrent declarative model

The following defines the syntax of a statement, ⟨s⟩ denotes a statement:

⟨s⟩ ::= skip
      | ⟨x⟩ = ⟨y⟩
      | ⟨x⟩ = ⟨v⟩
      | ⟨s₁⟩ ⟨s₂⟩
      | local ⟨x⟩ in ⟨s₁⟩ end
      | proc {⟨x⟩ ⟨y₁⟩ ... ⟨yₙ⟩ } ⟨s₁⟩ end
      | if ⟨x⟩ then ⟨s₁⟩ else ⟨s₂⟩ end
      | { ⟨x⟩ ⟨y₁⟩ ... ⟨yₙ⟩ }
      | case ⟨x⟩ of ⟨pattern⟩ then ⟨s₁⟩ else ⟨s₂⟩ end
      | thread ⟨s₁⟩ end

empty statement
variable-variable binding
variable-value binding
sequential composition
declaration
procedure introduction
conditional
procedure application
pattern matching
thread creation
The concurrent model

Top of Stack, Thread $i$ → $\text{thread } (s_1)_{\text{end}}, E$

ST

Single-assignment store
Basic Concepts

- The model allows multiple statements to execute “at the same time”?
- Imagine that these threads really execute in parallel, each has its own processor, but share the same memory
- Reading and writing different variables can be done simultaneously by different threads
- Reading the same variable can be done simultaneously
- Writing the same variable is done sequentially

Causal order

- In a sequential program all execution states are totally ordered
- In a concurrent program all execution states of a given thread is totally ordered
- The execution state of the concurrent program as a whole is **partially ordered**
Total order

- In a sequential program all execution states are totally ordered

Causal order in the declarative model

- In a concurrent program all execution states of a given thread is totally ordered
- The execution state of the concurrent program is partially ordered
Causal order in the declarative model

- fork a thread
- bind a dataflow variable
- synchronize on a dataflow variable
- computation step

Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there are multiple concurrent states
Example of nondeterminism

store

\begin{align*}
x \\
y = 5
\end{align*}

Thread 1

\begin{align*}
x = 1
\end{align*}

time

Thread 2

\begin{align*}
x = 3
\end{align*}

time

The thread that binds \( x \) first will continue, the other thread will raise an exception

Nondeterminism

- In the concurrent declarative model when there is only one binder for each dataflow variable, the nondeterminism is not observable on the store (i.e. the store develops to the same final results)
- This means for correctness we can ignore the concurrency
- Declarative concurrency
Scheduling

- The choice of which thread to execute next and for how long is done by a part of the system called the scheduler.

- A thread is *runnable* if its next statement to execute is not blocked on a dataflow variable, otherwise the thread is *suspended*.

Scheduling

- A scheduler is fair if it does not starve a runnable thread:
  - All runnable thread execute eventually.

- Fair scheduling make it easy to reason about programs.
- Otherwise some prefectly runnable program will never get its share.
Example of runnable thread

thread
  for I in 1..10000 do {Show 1} end
end
thread
  for I in 1..10000 do {Show 2} end
end

• This program will interleave the execution of two threads, one printing 1, and the other printing 2

• fair scheduler
Dataflow computation

- Threads suspend due to data availability in dataflow variables
- The `{Delay X}` primitive makes the thread suspend for X milliseconds, after that the thread is runnable

```plaintext
declare X
{Browse X}
local Y in
  thread {Delay 1000} Y = 10*10 end
  X = Y + 100*100
end
```

Concurrency Is Transparent

```plaintext
fun {CMap Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    thread {F X} end|{CMap Xr F}
  end
end
```
Concurrency Is Transparent

fun {CMap Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    thread {F X} end|{CMap Xr F}
  end
end

What happens:

declare F
{Browse {CMap [1 2 3 4] F}}
Concurrency Is Transparent

fun {CMap Xs F}
    case Xs
    of nil  then nil
    [] X|Xr then
        thread {F X} end|{CMap Xr F}
    end
end

- Browser shows [ _ _ _ _ ]
  - CMap computes the list skeleton
  - newly created threads suspend until F bound

What happens:

F = fun { $ X} X+1 end
Concurrency Is Transparent

fun \{CMap \ Xs \ F\}

case \ Xs
of \ nil \ then \ nil
[] \ X|Xr \ then
    thread \ \{F \ X\} \ end | \{CMap \ Xr \ F\}
end
end

- Browser shows [2 3 4 5]

Cheap concurrency and dataflow

- Declarative programs can easily be made concurrent
- Just use the thread statement where concurrent is needed
Cheap concurrency and dataflow

fun \{\text{Fib }X\} 
  \text{if } X==0 \text{ then } 0 
  \text{elseif } X==1 \text{ then } 1 
  \text{else} 
  \text{thread } \{\text{Fib }X-1\} \text{ end } + \{\text{Fib }X-2\} 
  \text{end} 
\text{end} 

Understanding why

fun \{\text{Fib }X\} 
  \text{if } \ldots \text{ then } 0 \text{ elseif } \ldots \text{ then } 1 
  \text{else} \ F1 \ F2 \ \text{in} 
  \begin{align*} 
  F1 &= \text{thread } \{\text{Fib }X-1\} \text{ end} \\
  F2 &= \{\text{Fib }X-2\} \\
  F1 + F2 \end{align*} 
\text{end} 

Execution of \{Fib 6\}

Fork a thread

Synchronize on result

Running thread

Fib

Panel | Options
---|---
| Threads | Memory | Problem Solving |
**Runtime**
Run: 1.74 s
Garbage Collection: 4.51 s
Copy: 0.00 s
Propagation: 0.00 s

**Threads**
Created: 124568
Runnable: 1
Producer ↔ Consumer

thread X={Produce} end
thread {Consume X} end

• Typically, what is produced will be put on a list that never ends (without \texttt{nil})

\texttt{stream}

• Consumer consumes as soon as producer produces

Example: Producer ↔ Consumer

\texttt{fun \{Produce N\}}
\hspace{1cm} N|{Produce N+1}
\texttt{end}
\texttt{proc \{Consume Xs\}}
\hspace{1cm} \texttt{case Xs of X|Xr then}
\hspace{2cm} \texttt{if X mod 1000 == 0 then}
\hspace{3cm} \{Browse X\}
\hspace{2cm} \texttt{end}
\hspace{2cm} \{Consume Xr\}
\texttt{end}
Stream Transducer

thread Xs={Produce} end
thread Ys={Transduce Xs} end
thread {Consume Ys} end

- Transducer
  - reads input stream
  - computes output stream
- Can be: filtering, mapping, …

Concurrent Streams

- Often used for simulation
  - analog circuits
  - digital circuits
- Lab assignment 4
  - streams used for simulation of analog circuits
  - simple circuits
  - lazy streams
Client ⇔ Server

- Similar to producer ⇔ consumer

Typical scenario:

- more clients than servers
- server has a fixed identity
- clients send messages to server
- server replies

Later: message sending

Fairness

- Essential that even though producer can always produce, consumer also gets a chance to run

- Threads are scheduled with fairness
  - if a thread is runnable, it eventually will run
Thread Scheduling

- More guarantees than just fairness
- Threads are given a time slice to run
  - approximately 10ms
  - when time slice is over: thread is preempted
  - next runnable thread is scheduled
- Can be even influenced by priorities
  - controls relative size of time slice [see book]

Summary

- Threads
  - suspend and resume automatically
  - controlled by variables
  - reminder: data-flow variables
  - cheap
  - execute fairly according to time-slice
- Pattern
  - producer ⇔ transducer ⇔ consumer
Demand Driven Execution

How to Control Producers?

- Producer should not produce more than needed

- Make consumer the stream producer
  - consumer produces skeleton, producer fills skeleton
  - difficult

- Use lazy streams: producer runs on request
Needed Variables

- Idea: start execution, when value for variable needed
  
  short: variable needed

- Value for variable needed…
  
  …a thread suspends on variable!

Triggers

- By-need triggers
  
  - a variable
  
  - a zero-argument function

- Trigger creation
  
  \[ X = \{ \text{ByNeed} \ F \} \]
The By-Need Protocol

- Suppose \((X, F)\) is a by-need trigger

- If \(X\) is needed,
  
  execute 
  \[
  \text{thread } X={F} \end
  \]
  
  delete trigger, \(X\) becomes a normal variable

Example: ByNeed

\[X=\{\text{ByNeed fun } \{\$\} \ 4 \ \text{end}\}\]

- Executing \{Browse X\}
  
  - shows <Future> (meaning not yet triggered)
  
  - Browser does not need variables!

- Executing \(Z=X+1\)
  
  - \(X\) is needed
  
  - thread \(T\) blocks (\(X\) is not yet bound)
  
  - new thread created that binds \(X\) to 4
  
  - thread \(T\) resumes and binds \(Z\) to 5
Lazy Functions

fun lazy {Produce N}
  N|{Produce N+1}
end

abbreviates

fun {Produce N}
  {ByNeed fun {$} N|{Produce N+1} end}
end

Lazy Production

fun lazy {Produce N}
  N|{Produce N+1}
end

- Intuitive understanding: function executes only, if its output is needed
Example: Lazy Production

fun lazy {Produce N}
    N|{Produce N+1}
end
declare Ns={Produce 0}
{Browse Ns}

- Shows again <Future>
  - Remember: the Browser does not need variables

execute _=Ns.1
- needs the variable Ns
- Browser now shows 0|<Future>
Example: Lazy Production

fun lazy {Produce N}
    N|{Produce N+1}
end
declare Ns={Produce 0}

- Execute _=Ns.2.2.1
  - needs the variable Ns.2.2
  - Browser now shows 0|1|2|<Future>

Everything Lazy!

- Not only producers, but also transducers can be made lazy

- Sketch
  - consumer needs variable
  - transducer is triggered, needs variable
  - producer is triggered
Lazy Example

fun lazy {Inc Xs}
    case Xs
    of X|Xr then X+1|{Inc Xr}
    end
end

declare Xs={Inc {Inc {Produce N}}}

Your Lab Assignment 4

- Producer
  - produce currencies
  - stream of current values

- Transducer
  - amplify streams
  - lazy transducer that scales stream values
Summary

- Demand-driven execution
  - execute computation, if variable needed
  - need is suspension by a thread
  - requested computation is run in new thread

- By-Need triggers

- Lazy functions

Outlook

- How can we capture threads in abstract machine?
  - have multiple semantic stacks
  - semantic stack = thread

- How about by-need?
  - have a store that contains trigger
  - rest is as we have sketched it
Important

- You must try the examples and go to the tutorial

- Things are simple and fun

Demand Driven Execution
Demand-driven Execution

- Let computations drive other computations
  - producer driven by consumer/transducer
  - module loader by thread needing module

- Variables control “demand” or “need”
  - variable needed: thread suspends on variable
  - by-need trigger:
    - variable
    - nullary function describing value to be computed
  - execution by newly created thread

Needed Variables

- Idea: start execution, when value for variable needed
  - short: variable needed

- Value for variable needed…
  - …a thread suspends on variable!
Triggers

- By-need triggers
  - a variable $X$
  - a zero-argument function $F$

- Trigger creation
  $X = \{\text{ByNeed } F\}$

The By-Need Protocol

- Suppose $(X, F)$ is a by-need trigger

- If $X$ is needed,

  execute thread $X = \{F\}$ end
  delete trigger, $X$ becomes a normal variable
Lazy Functions

fun lazy {Produce N}
  N|{Produce N+1}
end

abbreviates

fun {Produce N}
  {ByNeed fun {$} N|{Produce N+1} end}
end

Summary

• Demand-driven execution
  • execute computation, if variable needed
  • need is suspension by a thread
  • requested computation is run in new thread

• By-Need triggers

• Lazy functions
See You In Two Weeks!

See You Next Week!
Have a Nice Weekend