Declarative Concurrency

Reminder of Last Lecture

- Programming techniques
  - Types
  - Abstract data types
  - Haskell
  - Design methodology: functors + modules

Overview

- Declarative concurrency
- Mechanisms of concurrent program
- Streams
- Demand-driven execution
  - execute computation, if variable needed
  - needs suspension by a thread
  - requested computation is running in new thread
- By-Need triggers
- Lazy functions

The World is Concurrent!

- Concurrent programs
  - several activities execute simultaneously (concurrently)

- Most of the software used are 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 to execute
  - essential: how do concurrent programs interact?
- Concurrent programs can run faster on parallel machines (including clusters and cores)

Concurrency and Parallelism

- **Concurrency** is *logically simultaneous processing* which can also run on sequential machine.
- **Parallelism** is *physically simultaneous processing* and it involves multiple processing elements and/or independent device operations.
- A **computer cluster** is a group of connected computers that work together as a unit. One popular implementation is a cluster with nodes running Linux with support library (for parallelism).

Concurrent Programming is Difficult…

- This is the traditional belief
- The truth is: concurrency is very difficult…
  … if used with inappropriate tools and programming languages
- Particularly troublesome: *state* and *concurrency*

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) ... and incremental processing

Threads

Our First Concurrent 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

Our First Program

```plaintext
declare x0 x1 x2 x3
thread x1 = 1 + x0 end
thread x3 = x1 + x2 end
{Browse [x0 x1 x2 x3]}
```

- Both threads are suspended
  - x1 = 1 + x0 suspended; x0 unassigned
  - x3 = x1 + x2 suspended; x1, x2 unassigned
Our First Program

```plaintext
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 [4 5 X2 X3]

Our First Program

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

- Feeding  
  - X0 = 4

Second thread is still suspended  
  - Variable X2 is still not assigned
Our First Program

```oz
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 [4 5 2 7]

Threads

- A **thread** is simply an executing program.
- A program can have more than one thread.
- A thread is created by:
  ```oz
  thread 〈s〉 end
  ```
- Threads compute
  - independently
  - as soon as their statements can be executed
  - interact by binding variables in store

The Browser

- Browser is implemented in Oz as a thread.
- It also runs whenever browsed variables are bound
- It uses some extra functionality to look at unbound variables

Sequential Model

- Statements are executed sequentially from a single semantic stack
  - $w = a$
  - $z = \text{person}(\text{age: } y)$
  - $x$
  - $y = 42$
  - $u$
**Concurrent Model**

Multiple semantic stacks (threads)

Semantic Stack 1

Semantic Stack N

Single-assignment store

\[ w = a \]

\[ z = \text{person(age: } y) \]

\[ x \]

\[ y = 42 \]

\[ u \]

**Concurrent Declarative Model**

Kernel language extended with thread creation

\[ \langle s \rangle ::= \text{skip} \]

\[ \langle x \rangle = \langle y \rangle \]

\[ \langle x \rangle = \langle v \rangle \]

\[ \langle s_1 \rangle \langle s_2 \rangle \]

local \( \langle x \rangle \) in \( \langle s_1 \rangle \) end

\[ \text{proc} \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \langle s_1 \rangle \langle s_2 \rangle \end \]

if \( \langle x \rangle \) then \( \langle s_1 \rangle \) else \( \langle s_2 \rangle \) end

\[ \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \]

case \( \langle x \rangle \) of \( \langle \text{pattern} \rangle \) then \( \langle s_1 \rangle \) else \( \langle s_2 \rangle \) end

\[ \langle s_2 \rangle \text{end} \]

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 \( i \) → \( \text{thread } \langle s_1 \rangle \text{ end,E} \)

Single-assignment store

\[ \text{Top of Stack, Thread } i \rightarrow \text{ST} \]

**The Concurrent Model**

Top of Stack, Thread \( i \) → \( \text{ST} \)

Single-assignment store

\[ \text{Top of Stack, Thread } i \rightarrow \text{thread } \langle s_1 \rangle \text{ end,E} \]
Basic Concepts

- Model allows multiple statements to execute "simultaneously"?
- Can 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 concurrently.
- Writing to the same variable to be 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 are totally ordered.
- But, ... 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.
- Computation step: transition between two consecutive execution states.

Causal Order in the Declarative Model

- In a concurrent program all execution states of a given thread are totally ordered.
- The execution state of the concurrent program is partially ordered.

fork a thread

thread T1

thread T2

thread T3

computation step

sequential execution
Causal Order in the Declarative Model

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

Example of Nondeterminism

- store
- \( x \)
- \( y = 5 \)

<table>
<thead>
<tr>
<th>time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( x = 1 )</td>
<td>( x = 3 )</td>
</tr>
</tbody>
</table>

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

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

- If there is only one binder for each dataflow variable, nondeterminism is not *observable* on the store.
- That is the store has the same final results.
- Hence, for correctness we can ignore the concurrency
- This concept is known as "Declarative Concurrency".
Declarative concurrency

- **Declarative programming (Reminder):**
  - the output of a declarative program should be a mathematical function of its input.

- **Functional programming (Reminder):**
  - the program executes with some input values and when it terminates, it has returned some output values.

- **Data-driven concurrent model:** A concurrent program is **declarative** if all executions with a given set of inputs have one of two results:
  - (1) they all do not terminate or
  - (2) they all eventually reach partial termination and give results that are logically equivalent.

Partial Termination. Example

```haskell
fun {Double Xs}
case Xs of
  nil then nil
  [] X|Xr then 2*X|{Double Xr} end
end
Ys={Double Xs}
```

As long as input stream \( Xs \) grows, then output stream \( Ys \) grows too. The program never terminates.

However, if the input stream stops growing, then the program will eventually stop executing too.

The program does a *partial termination*.

Partial Termination. Examples

If the inputs are bound to some partial values, then the program will eventually end up in partial termination.

Also, the outputs will be bound to some partial values.

What is the relation of outputs in terms of inputs when we consider partial values?

Example:

\[ Xs = 1|2|3|Xr \rightarrow Ys \] will be bound to \( 2|4|6|_ \)

Having \( Xr = 4|5|Xr1 \), we get \( Ys \) bound to \( 2|4|6|8|10|_ \)

Making \( Xr1 = \text{nil} \), we get \( Ys \) bound to \( [2 4 6 8 10] \)

Logical Equivalence. Examples

What does store contents being “the same” means?

**Example 1:**
- Case 1: \( X=1 \) \( Y=X \)
- Case 2: \( Y=X \) \( X=1 \)

The store contents is the same for both cases

**Example 2:**
- Case 1: \( X=\text{foo}(Y W) \) \( Y=Z \)
- Case 2: \( X=\text{foo}(Z W) \) \( Y=Z \)

The store contents is the same for both cases
Logical Equivalence

- A set of store bindings is called a constraint.

- For each variable $x$ and constraint $c$, we define $values(x, c)$ to be the set of all possible values $x$ can have, given that $c$ holds.

  Example: $values(x, 2 < x < 8) = \{3, 4, 5, 6, 7\}$

Logical Equivalence. Example

- Example:
  - suppose that $x$, $y$, $z$, and $w$ are store variables.
  - the constraint $x = \text{foo}(y \ w) \land y = z$ is logically equivalent to the constraint $x = \text{foo}(z \ w) \land y = z$.
  - Reason: $y = z$ forces $y$ and $z$ to have the same set of possible values, so that $\text{foo}(y \ w)$ defines the same set of values as $\text{foo}(z \ w)$.

Logical Equivalence

- Two constraints $c_1$ and $c_2$ are logically equivalent if:
  1. they contain the same set of variables, and
  2. for each variable $x$, $values(x, c_1) = values(x, c_2)$.

Scheduling

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

- A thread is runnable if its next statement to execute is not blocked on a dataflow variable, otherwise the thread is suspended.
A scheduler is *fair* if it does not starve each runnable thread.
- All runnable threads execute eventually.

Fair scheduling makes it easier to reason about programs.

Otherwise some runnable programs will never get its turn for execution.

**Example of Runnable Threads**

```plaintext
thread
  for I in 1..10000 do {Browse 1} end
end

thread
  for I in 1..10000 do {Browse 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 when dataflow variables needed are not yet bound.
- `{Delay X}` primitive makes the thread suspends 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

Example: a concurrent map operation

```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

What happens:

```plaintext
declare F
{Browse {CMap [1 2 3 4] F}}
```

- Browser shows `[ _ _ _ _ ]`
- CMap computes the list skeleton
- newly created threads suspend until F becomes bound.
Concurrency is Transparent

- What happens:
  \[ F = \text{fun} \: \{ \$ \: X \} \: X+1 \: \text{end} \]

- Browser shows \([2 \: 3 \: 4 \: 5]\)

Cheap Concurrency and Dataflow

- Declarative programs can be easily made concurrent
- Just use the \text{thread} statement where concurrency is needed

Cheap Concurrency and Dataflow

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

Understanding why

\[
\text{fun} \: \{ \text{Fib} \: X \} \\
\quad \text{if} \: X==0 \: \text{then} \: 0 \ \text{elseif} \: X==1 \: \text{then} \: 1 \\
\quad \text{else} \: Y1 \: Y2 \: \text{in} \\
\quad \quad Y1 = \text{thread} \: \{ \text{Fib} \: X-1 \} \: \text{end} \\
\quad \quad Y2 = \{ \text{Fib} \: X-2 \} \\
\quad \quad Y1 + Y2 \\
\quad \text{end} \\
\text{end}
\]

\text{Dataflow dependency}
Streams

A most useful technique for declarative concurrent programming to use **streams** to communicate between threads.

- A **stream** is a potentially unbounded list of messages, i.e., it is a list whose tail is an unbound dataflow variable.
- A thread communicating through streams is a kind of “active object”, also called **stream object**.
- A sequence of stream objects each of which feeds the next is called a **pipeline**.
- **Deterministic stream programming**: each stream object always knows for each input where the next message will come from.
**Producer ↔ Consumer**

```plaintext
thread X={Produce} end
thread Result={Consume X} end
```

- Typically, what is produced will be put on a list that never ends (without nil), called **stream**
- **Consumer** (also called **sink**) consumes as soon as **producer** (also called **source**) produces

**Example: Producer ↔ Consumer**

```plaintext
fun {Produce N Limit}
  if N<Limit then
    N|{Produce N+1 Limit}
  else nil end
end

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

**Producer/Consumer Stream**

- **Producer**:
  - Xs=0|1|2|3|4|5|...
- **Consumer**:
  - Xs={Produce 0 Limit}
  - S={Consume Xs 0}

**Stream Transducer. Example**

```plaintext
thread Stream={Produce 0 1000} end
thread FilterResult={Filter Stream IsOdd} end
thread Result={Consume FilterResult 0} end
```

- **Transducer**: a stream which reads the producer’s output and computes a filtered stream for the consumer.
- Can be: filtering, mapping, ...

Advantages of pipeline:
- there is no need to wait the final value of the producer
- producer, transducer, and consumer are executed concurrently
Simple Pipeline

Producer \(\xrightarrow{Xs}\) Transducer \(\xrightarrow{Ys}\) Consumer

\[Ys = \{\text{Filter } Xs .\}\]

Concurrent Streams

- Often used for simulation
  - analog circuits
  - digital circuits (Section 4.3.5, pages 266-272)
  - 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
- See Next Lecture: 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 will eventually 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 influenced by priorities
  - high, medium, low
  - controls relative size of time slice (Sections 4.2.4-4.2.6)

Summary so far

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

Demand Driven Execution

How to Control Producers?

- Eager model: the producer decides when enough data has been sent
- Possible problem: producer should not produce more than needed
- One attempt: make consumer the driver
  - consumer produces stream skeleton
  - producer fills skeleton
Make Consumer be the Driver

fun \{DConsume ?Xs A Limit\}
    if Limit>0 then
        local X Xr in
            Xs=X|Xr \{DConsume Xr A+X Limit-1\}
    else A end
end
proc \{DProduce N Xs\}
    case Xs of X|Xr then
        X=N \{DProduce N+1 Xr\}
    end
end

Overall program :

local Xs S in
    thread \{DProduce 0 Xs\} end
    thread S=\{DConsume Xs 0 150000\} end
\{Browse S\}
end

Note that consumer controls how many elements are needed.

Bounded Buffer

- **Eager** – producer may run ahead

- **Demand-driven** – consumer in control but more complex execution.

- **Compromise**: Bounded Buffer
**Bounded Buffer Code**

```plaintext
proc {Buffer N Xs Ys}
  fun {Startup N ?Xs}
    if N==0 then Xs
    else Xr in Xs=_|Xr {Startup N-1 Xr} end
  end
  proc {AskLoop Ys ?Xs ?End}
    case Ys of Y|Yr then Xr End in
        Xs=Y|Xr % get element from buffer
        End=_|End2 % replenish the buffer
        {AskLoop Yr Xr End2}
    [] nil then End=nil end
  end
  End={Startup N Xs}
  in
  {AskLoop Ys Xs End}
end
```

**Lazy Streams**

- Better solution for demand-driven concurrency
  
  Use Lazy Streams
  
  That is consumer decides, so producer runs on request.

**Needed Variables**

- **Idea:**
  - start execution,
  - when value for variable needed
  - suspend on the variable

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

**Lazy Execution (Reminder)**

- Up to now the execution order of each thread follows textual order.
  - Each statement is executed in order strict order, whether or not its results are needed later.
- This execution scheme is called *eager execution*, or *supply-driven* execution
- Another execution order is to execute each statement only if its results are needed somewhere in the program
- This scheme is called *lazy evaluation*, or *demand-driven evaluation*
Lazy Execution. Reminder

declare
fun lazy {F1 X} 2*X end
fun {F2 Y} Y*Y end
B = {F1 3}
{Browse B} \rightarrow nothing (simply unbound B)
C = {F2 4}
{Browse C} \rightarrow display 16
A = B+C

- **F1** is a lazy function
- **B = {F1 3}** is executed only if its result is needed in **A = B+C**

Example

declare
fun lazy {F1 X} 2*X end
fun lazy {F2 Y} Y*Y end
B = {F1 3}
{Browse B} \% \rightarrow nothing (simply unbound B)
C = {F2 4}
{Browse C} \% \rightarrow nothing (simply unbound C)

- **F1 and F2** are now lazy functions
- **B = {F1 3} and C = {F2 4}** are executed only if their results are needed in an expression, like: **A = B+C**

Example

declare
fun lazy {F1 X} 2*X end
fun lazy {F2 Y} Y*Y end
B = {F1 X}
C = {F2 Y}
A = B+C

In lazy execution, an operation suspends until its result is needed
- Each suspended operation is triggered when another operation needs the value for its arguments
- In general, multiple suspended operations can start concurrently
Example II

- **In data-driven execution**, an operation suspends until the values of its arguments results are available.
- In general, the suspended computation can start concurrently.

\[
B = \{F_1 X\} \quad C = \{F_2 Y\}
\]

\[
A = B + C
\]

Triggers

- **A by-need trigger** is a pair \((F, X)\):
  - a zero-argument function \(F\)
  - a variable \(X\)

- **Trigger creation**
  \[X = \{\text{ByNeed } F\} \quad \text{or equivalently} \quad \{\text{ByNeed (proc } \{A\} \ A = \{F\} \text{ end) } X\}\]

- **If** \(X\) **is needed**, then \(X = \{\text{ByNeed } F\}\) **means**:
  - execute **thread** \(X = \{F\} \text{ end}\)
  - delete trigger, \(X\) becomes a normal variable

---

Example 1: ByNeed

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

- **Executing** \(\{\text{Browse } X\}\)
  - Shows: \(X\) (**meaning not yet triggered**)
  - Browse does not need the value of \(X\)

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

---

Example 2: ByNeed

\[
\text{declare fun } \{F_1 X\} \{\text{ByNeed fun } \{\$\} 2*X \text{ end} \end {end}
\]

\[
\text{fun } \{F_2 Y\} \{\text{ByNeed fun } \{\$\} Y*Y \text{ end} \end {end}
\]

\[
B = \{F_1 3\}
\]

\[
\{\text{Browse B}\} \quad \% \text{ simply display } B
\]

\[
C = \{F_2 4\}
\]

\[
\{\text{Browse C}\} \quad \% \text{ simply display } C
\]
Example 2: ByNeed

declare
fun {F1 X} {ByNeed fun {$} 2*X end} end
fun {F2 Y} {ByNeed fun {$} Y*Y end} end
B = {F1 3}
(Browse B) % display 6
C = {F2 4}
(Browse C) % display 16
A = B+C

Example 3: ByNeed

thread X={ByNeed fun {$} 3 end} end
thread Y={ByNeed fun {$} 4 end} end
thread Z=X+Y end

- Considering that each thread executes atomically, there are six possible executions.
- For lazy execution to be declarative, all of these executions must lead to equivalent stores.
- The addition will wait until the other two triggers are created, and these triggers will then be activated.

Lazy Functions

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

can be implemented with by-need triggers

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

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

- Shows again Ns
  - Remember: Browse does not need the values of the variables

---

Example: Lazy Production

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

- Execute `_=Ns.1`
  - needs the variable Ns
  - Browser now shows 0|_ or 0|<Future>

---

Example: Lazy Production

```scala
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|_
Lazy Transducer. Example

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

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

Global Summary

- Declarative concurrency
- Mechanisms of concurrent program
- Streams
- 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

Reading suggestions

- Chapter 4, Sections 4.1-4.5 from [van Roy, Haridi; 2004]
- Exercises 4.11.1-4.11.16 from [van Roy, Haridi; 2004]