Review of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language, Oz)
  - Message passing with active objects (Erlang, SALSA)
  - Atomic actions on shared state (Java)
- The atomic action approach is the most difficult, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
  - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, else message passing if you can get away with it.

Concurrency

- Some programs are best written as a set of activities that run independently (concurrent programs)
- Concurrency is essential for interaction with the external environment
- Examples includes GUI (Graphical User Interfaces), operating systems, web services
- Also programs that are written independently but interact only when needed (client-server, peer-to-peer applications)
- This lecture is about declarative concurrency, programs with no observable nondeterminism, the result is a function
- Independent procedures that execute on their pace and may communicate through shared dataflow variables

Overview

- Programming with threads
  - The model is augmented with threads
  - Programming techniques: stream communication, order-determining concurrency, coroutines, concurrent composition
- Lazy execution
  - Demand-driven computations, lazy streams, and list comprehensions
- Soft real-time programming
Concurrent declarative model

The following defines the syntax of a statement, \( s \) denotes a statement:

\[
\begin{align*}
\langle s \rangle & ::= \text{skip} \\
& | \langle x \rangle = \langle y \rangle \\
& | \langle x \rangle = \langle v \rangle \\
& | \langle s_1 \rangle \langle s_2 \rangle \\
& | \text{local} \langle x \rangle \text{in} \langle s \rangle \text{end} \\
& | \text{proc} \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \langle s \rangle \text{end} \\
& | \text{if} \langle x \rangle \text{then} \langle s \rangle \text{else} \langle s \rangle \text{end} \\
& | \text{case} \langle x \rangle \text{of} \langle \text{pattern} \rangle \text{then} \langle s \rangle \text{else} \langle s \rangle \text{end} \\
& | \text{thread} \langle a \rangle \text{end}
\end{align*}
\]

The concurrent model

- Top of Stack, Thread \( i \) → ST
- \( \langle b_i \rangle, E \)
- 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, as well as reading the same variable.
- Writing the same variable is done sequentially.
- The above view is in fact equivalent to an interleaveing execution: a totally ordered sequence of computation steps, where threads take turns doing one or more steps in sequence.

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.
- 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 are totally ordered.
- The execution state of the concurrent program is partially ordered.

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 is concurrent access to shared state.

Example of nondeterminism

- The thread that binds x first will continue, the other thread will raise an exception.

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.
- A scheduler is fair if it does not starve a runnable thread, i.e. all runnable threads eventually execute.
- Fair scheduling makes it easy to reason about programs and program composition.
- Otherwise some correct program (in isolation) may never get processing time when composed with other programs.
The semantics

- In the sequential model we had:
  \[(ST, \sigma)\]
  - \(ST\) is a stack of semantic statements
  - \(\sigma\) is the single assignment store

- In the concurrent model we have:
  \[(MST, \sigma)\]
  - \(MST\) is a (multi)set of stacks of semantic statements
  - \(\sigma\) is the single assignment store

The initial execution state

- Statement
- Stack: \([\{(s, \mathcal{O})\} \}, \mathcal{O}\]
- Multiset
- Store

Execution (the scheduler)

- At each step, one runnable semantic stack is selected from MST (the multiset of stacks), call it ST, s.t.
  \(MST = ST \cup MST'\)
- Assume the current store is \(\sigma\), one computation step is done that transforms ST to ST' and \(\sigma\) to \(\sigma'\)
- The total computation state is transformed from \((MST, \sigma)\) to \((ST' \cup MST', \sigma')\)
- Which stack is selected, and how many steps are taken is the task of the scheduler, a good scheduler should be fair, i.e., each runnable "thread" will eventually be selected
- The computation stops when there are no runnable stacks

Example of runnable threads

- This program will interleave the execution of two threads, one printing 1, and the other printing 2
- We assume a fair scheduler

Dataflow computation

- Threads suspend on data unavailability in dataflow variables
- The \([\text{Delay } X]\) primitive makes the thread suspends for \(X\) milliseconds, after that, the thread is runnable

Illustrating dataflow computation

- Enter incrementally the values of \(X0\) to \(X3\)
- When \(X0\) is bound the thread will compute \(Y0=X0+1\), and will suspend again until \(X1\) is bound
**Concurrent Map**

fun (Map Xs F)
case Xs
of nil then nil
([] X|Xr) then thread (F X) end
end

- This will fork a thread for each individual element in the input list
- Each thread will run only if both the element X and the procedure F is known

**Concurrent Map Function**

fun (Map Xs F)
case Xs
of nil then nil
([] X|Xr) then thread (F X) end
end

- What this looks like in the kernel language:

```
proc (Map Xs F Rs)
case Xs
of nil then Rs = nil
([] X|Xr) then R Rr in
Rs = R|Rr
thread R = {F X} end
{Map Xr F Rr}
end
end
```

**How does it work?**

- If we enter the following statements:
  ```
declare F X Y Z
{Browse thread (Map X F) end}
```
- A thread executing Map is created.
- It will suspend immediately in the case-statement because X is unbound.
- If we thereafter enter the following statements:
  ```
X = 1|2|Y
fun (F X) X*X end
```
- The main thread will traverse the list creating two threads for the first two arguments of the list:
  ```
thread {F 1} end,
and thread {F 2} end
```
- After entering:
  ```
Y = 3|Z
Z = nil
```
- The program will complete the computation of the main thread and the newly created thread:
  ```
thread {F 3} end
```
- Resulting in the final list 
  ```
[1 4 9]
```

**Simple concurrency with dataflow**

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

```
fun (Fib X)
if X=<2 then 1
else
    thread (Fib X-1) end + (Fib X-2)
end
```

**Understanding why**

```
fun (Fib X)
if X=<2 then 1
else
    thread (Fib X-1) end = (Fib X-2)
end
```

(Dataflow dependency)
Streams

- A stream is a sequence of messages
- A stream is a First-In First-Out (FIFO) channel
- The producer augments the stream with new messages, and the consumer reads the messages, one by one.

```
producer X1 X2 X3 X4 X5
```

Stream Communication I

- The data-flow property of Oz easily enables writing threads that communicate through streams in a producer-consumer pattern.
- A stream is a list that is created incrementally by one thread (the producer) and subsequently consumed by one or more threads (the consumers).
- The consumers consume the same elements of the stream.

Stream Communication II

- **Producer**, produces incrementally the elements
- **Transducer(s)**, transform(s) the elements of the stream
- **Consumer**, accumulates the results

Stream communication patterns

- The producer, transducers, and the consumer can, in general, be described by certain program patterns
- We show various patterns
Producer

fun (Producer State)
  if (More State) then
    X = (Produce State) in
    X | (Producer (Transform State))
  else nil end
end

- The definition of More, Produce, and Transform is problem dependent
- State could be multiple arguments
- The above definition is not a complete program!

Example Producer

fun (Generate N Limit)
  if N=<Limit then
    N | (Generate N+1 Limit)
  else nil end
end

- The State is the two arguments N and Limit
- The predicate More is the condition N=<Limit
- The predicate Produce is the identity function on N
- The Transform function (N,Limit) ⇒ (N+1,Limit)

Consumer Pattern

fun (Consumer State InStream)
  case InStream of nil then (Final State) in
    NextState = (Consume X State) in
    (Consumer NextState RestInStream)
  end

- Final and Consume are problem dependent

Example Consumer

fun (Sum A Xs)
  case Xs of nil then A
  else Xr then (Sum A+X Xr) end
end

- The State is A
- Final is just the identity function on State
- Consume takes X and State ⇒ X + State

Transducer Pattern 1

fun (Transducer State InStream)
  case InStream of nil then nil
  else X | RestInStream then
    NextStateTX = (Transform X State) in
    TX | (Transducer NextState RestInStream)
  end

- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

Transducer Pattern 2

fun (Transducer State InStream)
  case InStream of nil then nil
  else X | RestInStream then
    NextStateTX = (Transform X State) in
    TX | (Transducer NextState RestInStream)
  end

- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream
Example Transducer

fun \( \text{Filter } Xs \ F \) case \( Xs \) of \( \text{nil} \) then \( \text{nil} \) \[ \] \( X | Xr \) then if \( \{ F \ X \} \) then \( X | \{ \text{Filter } Xr \ F \} \) else \( \{ \text{Filter } Xr \ F \} \) end end end

Larger example: The sieve of Eratosthenes

- Produces prime numbers
- It takes a stream \( 2 \ldots N \), peals off 2 from the rest of the stream
- Delivers the rest to the next sieve

Sieve

fun \( \{ \text{Sieve } Xs \} \) case \( Xs \) of \( \text{nil} \) then \( \text{nil} \) \[ \] \( X | Xr \) then \( \text{local } Xs \ Ys \text{ in} \) thread \( Ys = \{ \text{Filter } Xr \ \text{fun \( \$ Y \) \text{mod} \( X \) \( = \text{0} \) \text{end} \text{end} \text{end} \text{X} | \{ \text{Sieve } Ys \} \} \) end end • The program forks a filter thread on each sieve call

Example call

local \( Xs \ Ys \text{ in} \) thread \( Xs = \{ \text{Generate } 2 \ 100000 \} \text{ end} \text{thread} \ Ys = \{ \text{Sieve } Xs \} \text{ end} \text{thread for } Y \text{ in } Ys \text{ do } \{ \text{Show } Y \} \text{ end end} \)

Limitation of eager stream processing Streams

- The producer might be much faster than the consumer
- This will produce a large intermediate stream that requires potentially unbounded memory storage

Solutions

There are three alternatives:
1. Play with the speed of the different threads, i.e. play with the scheduler to make the producer slower
2. Create a bounded buffer, say of size \( N \), so that the producer waits automatically when the buffer is full
3. Use demand-driven approach, where the consumer activates the producer when it needs a new element (lazy evaluation)

- The last two approaches introduce the notion of flow-control between concurrent activities (very common)
Coroutines I

- Languages that do not support concurrent threads might instead support a notion called coroutining.
- A coroutine is a nonpreemptive thread (sequence of instructions), there is no scheduler.
- Switching between threads is the programmer’s responsibility.

Coroutines II, Comparison

- **Procedures**: one sequence of instructions, program transfers explicitly when terminated it returns to the caller.
- **Coroutines**: New sequences of instructions, programs explicitly do all the scheduling, by spawn, suspend and resume.

Time

- In concurrent computation one would like to handle time.
- `proc (Time.delay T)` – The running thread suspends for T milliseconds.
- `proc (Time.alarm T U)` – Immediately creates its own thread, and binds U to `unit` after T milliseconds.

Example

```plaintext
local proc {Ping N}
for I in 1..N do
  {Delay 500} {Browse ping}
end
{Browse 'ping terminate'}
end

proc {Pong N}
for I in 1..N do
  {Delay 600} {Browse pong}
end
{Browse 'pong terminate'}
end

in ...
local ...
in
  {Browse 'game started'}
thread (Ping 1000)
thread (Pong 1000)
end

Concurrent control abstraction

- We have seen how threads are forked by `thread ... end`.
- A natural question to ask is: how can we join threads?
Termination detection

- This is a special case of detecting termination of multiple threads, and making another thread wait on that event.
- The general scheme is quite easy because of dataflow variables:

  ```
  thread〈S1〉 X1 = unit end
  thread〈S2〉 X2 = X1 end
  ...
  thread〈Sn〉 Xn = Xn-1 end
  
  {Wait Xn} % Continue main thread
  
  When all threads terminate the variables X1 … XN will be merged together labeling a single box that contains the value unit.
  
  {Wait XN} suspends the main thread until XN is bound.
  ```

Concurrent Composition

- Takes a single argument that is a list of nullary procedures.
- When it is executed, the procedures are forked concurrently. The next statement is executed only when all procedures in the list terminate.

Example

```plaintext
local proc {Conc1 Ps I O}
  case Ps of
    P|Pr then
      M in thread〈P〉 M = I end
      {Conc1 Pr M O}
    nil then
      O = I end
    end
  end
in proc {Conc Ps}
  X in {Conc1 Ps unit X} (Wait X) end
in ...
end
```

Futures

- A future is a read-only capability of a single-assignment variable. For example to create a future of the variable X we perform the operation !! to create a future Y: Y = !!X
- A thread trying to use the value of a future, e.g. using Y, will suspend until the variable of the future, e.g. X, gets bound.
- One way to execute a procedure lazily, i.e. in a demand-driven manner, is to use the operation {ByNeed +P?F}.
- ByNeed takes a zero-argument function P, and returns a future F. When a thread tries to access the value of F, the function (P) is called, and its result is bound to F.
- This allows us to perform demand-driven computations in a straightforward manner.

Example

```plaintext
local proc {Ping N}
  for I in 1..N do
    {Delay 500} {Browse ping} end
  {Browse 'ping terminate'}
end proc
{Pong N}
  for I in 1..N do
    {Delay 600} {Browse pong} end
  {Browse 'pong terminate'}
end in ...
end
```

```plaintext
local ...
in {Browse 'game started'}
  {Conc proc {$} {Ping 1000} end proc {$} {Pong 1000} end
  {Browse 'game terminated'}} end
```

- we will observe that Y becomes a future, i.e. we will see Y<Future> in the Browser.
- If we try to access the value of Y, it will get bound to 1.
- One way to access Y is by perform the operation {Wait Y} which triggers the producing procedure.
Thread Priority and Real Time

- Try to run the program using the following statement:
  ```
  {Sum 0 thread {Generate 0 100000000} end}
  ```
- Switch on the panel and observe the memory behavior of the program.
- You will quickly notice that this program does not behave well.
- The reason has to do with the asynchronous message passing. If the producer sends messages i.e. create new elements in the stream, in a faster rate than the consumer can consume, increasingly more buffering will be needed until the system starts to break down.
- One possible solution is to control experimentally the rate of thread execution so that the consumers get a larger time-slice than the producers do.

Priorities

- There are three priority levels:
  - high,
  - medium, and
  - low (the default)
- A priority level determines how often a runnable thread is allocated a time slice.
  - In Oz, a high priority thread cannot starve a low priority one. Priority determines only how large piece of the processor-cake a thread can get.
  - Each thread has a unique name. To get the name of the current thread the procedure `Thread.this/1` is called.
  - Having a reference to a thread, by using its name, enables operations on threads such as:
    - terminating a thread, or
    - raising an exception in a thread.
- Thread operations are defined the standard module `Thread`

Thread priority and thread control

```prolog
fun {Thread.state T} % returns thread state
  proc {Thread.injectException T E} % exception E injected into thread
  fun {Thread.this} % returns 1st class reference to thread
  proc {Thread.setPriority T P} % P is high, medium or low
  proc {Thread.setThisPriority P} % as above on current thread
  fun {Property.get priorities} % get priority ratios
  proc {Property.put priorities(high:H medium:M)}
```

The program with priorities

```prolog
local L in
  {Property.put priorities p(high:10 medium:10)}
thread
  {Thread.setThisPriority low}
  {Generate 0 100000000}
end
thread
  {Thread.setThisPriority high}
  {Sum 0 L}
end
```

Exercises

85. SALSA asynchronous message passing enables to tag messages with properties: priority, delay, and waitfor. Compare these mechanisms with Oz thread priorities, time delays and alarms, and futures.
86. How do SALSA tokens relate to Oz dataflow variables and futures?
87. What is the difference between multiple thread termination detection in Oz and join blocks in SALSA?
88. VRH Exercise 4.11.3 (page 339) - Compare the sequential and concurrent execution performance of equivalent SALSA programs.
89. VRH Exercise 4.11.5 (page 339)