LINFO1131
Advanced Programming Language Concepts

Course slides

“Programs should always be declarative except where they interact with the real world”

December 2023

Peter Van Roy
École Polytechnique de Louvain (Louvain Engineering School)
Université catholique de Louvain
B-1348 Louvain-la-Neuve
Belgium

Lecture	Topic	Page
1	Introduction and refresher	3
2 & 3	Lazy evaluation and declarative programming	25
4	Advanced declarative algorithm design	67
5	Limitations of declarative programming	93
6	Data abstractions	110
7	Message passing and multi-agent programming	126
	Oz program examples for Lecture 7: Port objects, active objects, Flavius Josephus problem, lift control system	149
8	Robust multi-agent programming in Erlang	156
9	Shared-state concurrency: introduction, locks, and tuple spaces	190
	Oz program examples for Lecture 9: Concurrent queue, locks, tuple space	210
10	Shared-state concurrency: monitors	215
	Oz program examples for Lecture 10: Monitor implementation, bounded buffer	227
11	Shared-state concurrency: transactions	232
	Oz program examples for Lecture 11: Transaction manager, example transactions	257

LINFO1131
Concurrent programming concepts

Lecture 1:
Introduction and refresher

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Course overview
  - Course organization
  - Course content
- Refresher of the previous course LINFO1104
  - Functional programming
  - Recursion and invariant programming
  - Higher-order programming
  - Symbolic programming
  - Data abstraction
  - Deterministic dataflow
  - Nondeterminism

2

Course organization

- Organization
  - **Weekly lectures**: presence strongly recommended
  - **Weekly labs**: presence strongly recommended; 1 point bonus if present during all labs
  - **Moodle**: all slides and announcements will be there
- Grading
  - **Midterm**: optional; around week 7; corresponds to 5 points on exam
  - **Project**: mandatory; groups of 2 students; last half of quadrimester; must be done during quadrimester
  - **Exam**: 5 points corresponds to midterm (max of both), 10 points for rest of course

3

Software

- Oz language:
  - Mozart 2 system
  - www.mozart2.org
- Erlang language:
  - Erlang/OTP 25 system
  - www.erlang.org

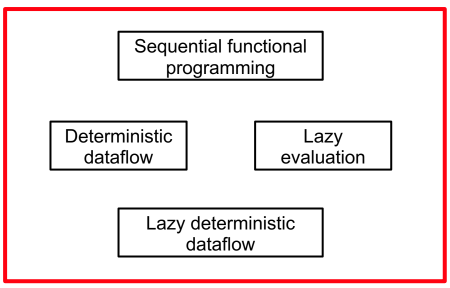
4

Three main themes

- Understanding concurrent programming
  - This is very hard in general
  - We will see how it can be made easy
- Understanding declarative programming
  - This is the easiest and best way to program
  - We will see how to use it as much as possible
- Understanding nondeclarative programming
  - This is quite hard but it is sometimes needed
  - We will see how to make it reasonably easy

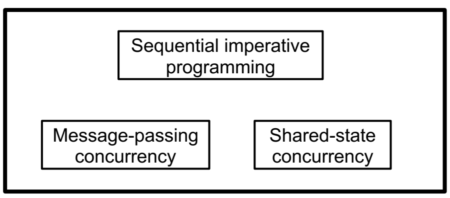
5

Declarative programming



Declarative programming

Functional programming
Higher-order programming
Lazy and concurrent programming
Semantics of declarative concurrency
Advanced declarative algorithm design



Nondeclarative programming

Limitations of declarative programming
Minimizing nondeterminism
Multi-agent programming
Fault tolerance and supervisors, Erlang
Locks, monitors, and transactions

6

Approximate course schedule

- S1: Introduction and refresher

Advanced declarative programming
- S2-S3: Lazy evaluation and lazy deterministic dataflow
- S3: Declarative concurrency
- S4-S5: Advanced declarative algorithm design
- S5: Limits of declarative programming

SMART week
- S7: Midterm

Advanced nondeclarative programming
- S6: Data abstraction
- S8: Multi-agent programming
- S9-S10: Robust multi-agent programming in Erlang
- S11: Shared-state concurrency, locks and tuple spaces
- S12: Monitors
- S13: Transactions

- S14: Course review

7

Refresher of previous course (LINFO1104)

8

Refresher

- In the rest of today's lecture, we recapitulate the main concepts needed for LINFO1131
- These concepts are taught in the previous course LINFO1104
  - If anything is not clear, please look it up!
  - Starting with next week's lecture, I will assume that all of this is *perfectly understood*

9

Functional programming

10

Functional programming

- It is the foundation of all programming
  - Higher-order programming is the foundation of all data abstraction
  - It is the best paradigm for testing, maintenance, and proving correctness
  - It is more and more being used, e.g., cloud analytics based on MapReduce and its successors
- In this course we will push it as far as we can
  - For concurrency: deterministic dataflow
  - For efficiency: advanced algorithm design

11

Invariant programming

12

Recursive functions

- In other words, loops!
  - Tail recursion = while loop
- Factorial example

  Invariant: n! = i! x a

  where: n is a constant

  i decreases

  a increases
  - "Principle of communicating vases"
- We give code in C and Oz…

13

Factorial example

- C code:

```
int fact(int n) {
    int a=1;
    int i=n;
    while (i>0) {
        a=i*a;
        i=i-1;
    }
    return a;
}
x=fact(10);
```

- Oz code:

```
fun {Fact A I}
   if (I>0) then
      {Fact I*A I-1}
   else A end
end
X={Fact 1 10}
```

14

Higher-order programming

15

Higher-order programming

- A function is a value in the language
- Key concept for defining data abstractions
- Example: (function as output)

```
fun {AddN N}
   fun {$ X} X+N end
end
Add5={AddN 5}
Add10={AddN 10}
```

- What is the order of AddN?

16

Closures

- Synonyms:
  - Closure ("fermeture")
  - Lexically scoped closure
  - Procedure value
- Closure in memory is code+environment:

```
a1 = (proc {$ X R} R=X+N end, {N→n}),
a2 = (proc ($ X R} R=X+N end, {N→m}),
n=5,
m=10
```

17

Map function

- Example with function as argument:

```
fun {Map L F}
   case L of nil then
      nil
   [] H|T then
      {F H}|{Map T F}
   end
end
```

- This function is tail-recursive! Why?

18

More abilities of higher-order

- Higher-order programming can do many things:
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator (Fold)
  - Encapsulation
  - Delayed execution

19

Symbolic programming

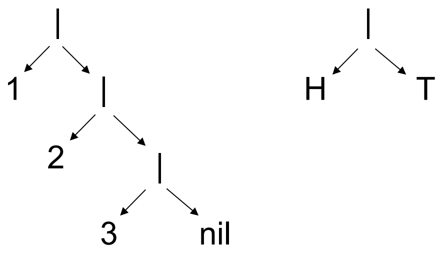
20

List data structure

- A list is a recursive data structure
  <L> ::= nil | <E> ‘|’ <L>
  - Recursive list function follows data definition
- Lists are a ubiquitous data structure
  - Languages give them syntactic supporrt
  - a|b|c|nil, [a b c], ‘|’(1:a 2:’|’(1:b 2:’|’(1:c 2:nil)

21

Pattern matching



- **case** L **of** H|T **then** … **end**
  - The case instruction matches a pattern, which is a shape of the matched structure
  - Patterns can match or fail to match

22

Fold operation

- The fold operation abstracts an accumulator
  - It uses higher-order programming to encapsulate an accumulator inside its definition
- Given a list L=[a_0 a_1 … a_{n-1}]
  - Define the following function:
    - s = (···((u f a_0) f a_1) · · ·) f a_{n-1})
    - s = fold(l f u)
- What is the Oz definition of fold?

23

Fold definition

- Recursive function with accumulator:

```
fun {FoldL L F U}
   case L of nil then U
   [] H|T then {FoldL T F {F U H}}
   end
end
```

- How can we use FoldL to compute the sum of elements of a list?
  - Is FoldL a tail-recursive function?

24

Data abstraction

25

Data abstraction

- A data abstraction encapsulates part of a program so that it can only be accessed through an interface
  - The interface can only be used through predefined rules
- Advantages of data abstraction
  - Guarantee that the abstraction will work
  - Reduction of complexity
  - Developing large programs with teams

26

Defining a data abstraction

- Data abstractions are defined with two concepts
  - Lexical scoping
  - Higher-order programming
- There are two main kinds of data abstractions
  - Objects
  - Abstract data types

27

A stack as abstract data type

- We define a stack as an ADT: (this is a semantic definition)

```
local Wrap Unwrap in
     {NewWrapper Wrap Unwrap}

     fun {NewStack} {Wrap nil} end
     fun {Push W X} {Wrap X|{Unwrap W}} end
     fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
     fun {IsEmpty W} {Unwrap W}==nil end
end
```

- How does this work?
- Look at the Push function: it first calls {Unwrap W}, which returns a stack value S, then it builds X|S, and finally it calls {Wrap X|S} to return a protected result
- Wrap and Unwrap are hidden from the rest of the program (static scoping)

28

A stack as an object

- We define a stack as an object: (this is a semantic definition)

```
fun {NewStack}
    C={NewCell nil}
    proc {Push X} C:=X|@C end
    proc {Pop X} S=@C in C:=S.2 X=S.1 end
    proc {IsEmpty B} B=(@C==nil) end
in
    proc {$ M}
       case M of push(X) then {Push X}
       [] pop(X) then {Pop X}
       [] isEmpty(B) then {IsEmpty B} end
    end
end
```

- How does this work?
- The object is represented by a one-argument procedure that does procedure dispatching: a case statement chooses the operation to execute
- Encapsulation is enforced by hiding the cell with static scoping

Special syntax for objects

- Most widely used languages provide a special syntax for objects
  - The semantics is the same
- This syntax guarantees that a programmer defines the object in the right way!
  - It also makes the program more readable
- Here is the Oz syntax for the stack object defined in the previous slide

```
class Stack
   attr c
   meth init
       c:=nil
   end
   meth push(X)
       c:=X|@c
   end
   meth pop(X)
       S=@c in X:=S.1 c:=S.2
   end
   meth isEmpty(B)
       B=(@c==nil)
   end
end

S={New Stack init}
```

Deterministic dataflow

31

Deterministic dataflow

- Concurrency = multiple activities executing simultaneously
- We provide a language operation to create a new concurrent activity:

  thread <s> **end**

  - Instruction <s> is executed independently of other activities

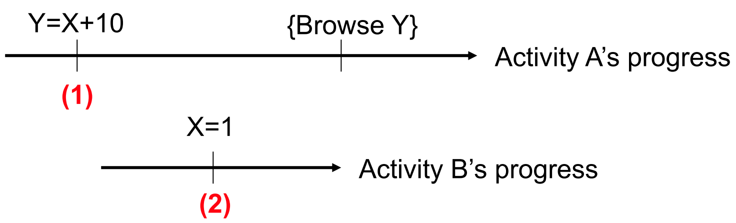
32

Dataflow concurrency

- Functional programming with concurrency
  - Unbound variables and threads
- Threads communicate with each other through shared variables
  - One thread binds the variable and one thread reads the variable
  - **thread** X=1 **end**
    thread Y=X+10 {Browse Y} **end**
  - This program always displays 11! It does not matter in which order the threads execute. Why not?

33

Always the same result!



- Activity A waits patiently at point **(1)** just before the addition
- When activity B binds X=1 at point **(2)**, then activity A can continue
- If activity B binds X=1 before activity A reaches point **(1)**, then activity A does not have to wait

34

Concurrent pipeline

- A *stream* is a list that ends in an unbound variable, which can be used as a channel
- An *agent* is a concurrent activity that reads and writes streams
- All list functions can be used as agents
  - Because list functions are tail-recursive, the agent will have constant stack size. Why are list functions tail-recursive?
  - All functional programming techniques can be used in deterministic dataflow

- Producer:

```
fun {Prod N}
   N|{Prod N+1}
end
```

- Consumer:

```
fun {Cons S}
   case S of H|T then
      {Browse H}
      {Cons T}
   end
end
```

- Pipeline:

```
thread S={Prod 1} end
thread {Cons S} end
```

35

Nondeterminism and the semantics of concurrency

36

Semantics of concurrency

- Each thread executes as a sequence of steps:
  T_1: $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow \cdots$
  T_2: $e'_0 \rightarrow e'_1 \rightarrow e'_2 \rightarrow \cdots \rightarrow \cdots$
- All threads are executed on one processor
  - The processor is sequential; it executes one instruction sequence (we assume a single core!)
  - Thread executions are *interleaved* on the processor
  - The *scheduler* chooses which thread to execute
  - The scheduler must be *fair* ; for efficiency and practicality, real systems also add *time slices* and *priorities*

37

Nondeterminism

- A program is *nondeterministic* when it executes an operation that is chosen external to the program
  - Scheduler choices are an example of nondeterminism
- Nondeterminism is inherent to any concurrent program
  - Threads are *independent* by definition, so the scheduler must make its choice outside of the control of the programmer
- Nondeterminism is inherent to any program that interacts with the real world
  - In the general case, real world events can happen at any time and must be handled when they occur
  - To interact with the real world, the program must be nondeterministic

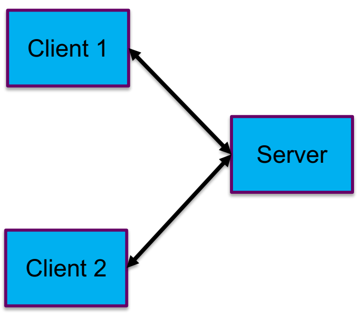
38

Client/server example

- A client/server is a typical example of a program that interacts with the real world and therefore makes choices external to the program
- The nondeterminism is a direct consequence of the client/server specification:
  - When a client makes a request to the server, the server must respond in a timely fashion that depends only on the network travel time and computation time of the request
  - Why is this?

39

Client/server application



- Specification:
  - When a client makes a request to the server, the server must respond in a timely fashion that depends only on the network travel time and computation time of the request
- Therefore the order of the requests cannot be determined in advance because it depends on precise timing of messages and computations
  - The order of message arrival at the server is determined external to the program!
- The whole client/server application is therefore nondeterministic, even if all the other code is purely declarative

40

40

Living with nondeterminism

- A nondeterministic program is *much harder* to develop and debug than a deterministic program
  - The program must work correctly for all possible nondeterministic choices
  - In general, there are very many such choices and they can happen at any time during the execution
    - Testing is very hard: we need to simulate all these choices!
- So what can we do?
  - This is one of the main themes of the course!
  - How to live with nondeterminism

How to live with nondeterminism

- A main theme of the course
- Two solutions:
  - **Deterministic dataflow**: this paradigm solves the problem for us. Even though the scheduler is doing nondeterministic choices, the results of the program are always the same. The nondeterminism is hidden because of the strong properties of functional programming.
  - **Nondeclarative programs**: the programmer has to solve the problem. The way to do this is by defining the right abstractions. We will see this later!

Conclusions

43

Conclusions

- LINFO1131 continues the story of LINFO1104
  - Concurrent programming
  - Declarative programming
  - Nondeclarative programming
- In today's lecture we explained the goals and organization of LINFO1131
- We recalled the most important concepts of LINFO1104
  - Please refer back to that course if anything is not clear
- Next week's lecture will be on lazy evaluation

44

LINFO1131
Concurrent programming concepts

Lectures 2 and 3: Lazy evaluation and declarative programming

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Introduction to lazy evaluation
  - Semantics based on dataflow
- Lazy streams
  - Producer-consumer in four paradigms
  - Infinite lists
  - Hamming problem
- Lazy suspensions
  - Graphical representation of lazy evaluation
- Lazy deterministic dataflow
  - Bounded buffer
- Lazy quicksort
  - Inventing an incremental algorithm
- What is declarative programming?
  - Partial termination
  - Equivalent stores
  - Introduction to first-order logic
  - Definition of declarative programming
  - Failure confinement
- Table of declarative paradigms
- Conclusions

2

Introduction to lazy evaluation

3

Introduction to lazy evaluation

- A lazy program is a functional program that executes in “by-need” fashion
  - Nothing is computed until it is “needed”
- Here is a simple example:

```
fun lazy {LazyAdd X Y}
    X+Y
end
S={LazyAdd 10 20}
{Browse S}
```

- Nothing is executed until S is needed:

```
% Displaying the addition S+100 needs S:
{Browse S+100}
```

4

Semantics of lazy evaluation

5

Semantics of LazyAdd

- How does LazyAdd work?
  - Semantics of a program is defined by translation into kernel language
  - We will define what “needing a value” means
- We translate into kernel language:

```
proc {LazyAdd X Y R}
    thread
        {WaitNeeded R} R=X+Y
    end
end
```

- The {WaitNeeded R} waits until another thread needs R to continue
  - More precisely, it waits until another thread does {Wait R}
  - This is part of dataflow execution…

6

Dataflow semantics

- To understand WaitNeeded, we first recall how dataflow execution works
- Given any expression:
  S=X+Y
- This is translated as:

```
local V in
    {Wait X}
    {Wait Y}
    {PrimitiveAdd X Y V}
    {Bind S V}
end
```

- This gives a dataflow execution:
  - {Wait X} suspends until X is bound
  - {Bind X V} binds X to V
- Programmer-accessible operations are defined using Wait, Bind, and a primitive operation:
  - Arithmetic, boolean expressions
  - Case statements
  - Any operation with an input
    - Function call {F X} where F must be bound to a function value
    - Dot operation R.name where R must be bound to a record

{WaitNeeded X} suspends until another thread does {Wait X}

7

Another example

- We use WaitNeeded directly:

```
declare X in
{WaitNeeded X}
X=100
```

- This displays an unbound variable:
  {Browse X}
- This displays 100 twice (!):
  {Browse X+0}

8

General translation scheme

- Given any lazy function:

```
fun lazy {F X1 … Xn}
     <expr>
end
```

- This is translated into:

```
proc {F X1 … Xn R}
     thread
        {WaitNeeded R} R=<expr>
     end
end
```

- This translation gives the semantics, not the implementation!
  - A compiler is free to optimize it while respecting the semantics

9

Producer-consumer in four paradigms

10

Producer-consumer pipeline

- We give the code of a simple producer-consumer pipeline
  - We will run the code in four different functional paradigms
  - All four paradigms are declarative and end up with the same result
  - But the result appears in four different ways
- Technically we are just taking advantage of the Church-Rosser theorem
  - All reduction orders of a lambda expression give the same result
  - Also called confluence

```
fun {Prod L H}
   {Delay 1000} % Wait 1000 ms
   if L>H then nil
   else L|{Prod L+1 H}
   end
end

fun {Cons S Acc}
   case S of H|T then
      Acc+H|{Cons T Acc+H}
   [] nil then nil
   end
end
```

11

Four functional paradigms

As seen earlier

1. Sequential functional programming
   - No single assignment
   - Traditional functional languages (Lisp, Scheme, ML, OCaml)
2. Sequential FP with single assignment
   - Default execution of functions in Oz
3. Deterministic dataflow
   - Adds threads and dataflow synchronization
   - Multi-agent programming with declarative agents

New!

4. Lazy evaluation (introduced in this lecture!)
   - Adds by-need synchronization
   - Lazy functional languages (Haskell, Miranda)

12

1. Sequential FP (no single assignment)

- We generate 10 elements
  - This is traditional FP with no single assignment
  - Nothing is displayed until after 10 seconds
    - S1 and S2 both displayed at once after 10 seconds
  - Both S1 and S2 are created as a batch execution
    - Why?

```
% Prod2 uses no unbound variables
fun {Prod2 L H}
   {Delay 1000} % Wait 1000 ms
   if L>H then nil
   else S in
      S={Prod2 L+1 H}
      L|S % Both L and S are bound
   end
end
```

```
declare S1 S2 in
{Browse S1}
{Browse S2}
S1={Prod2 1 10}
S2={Cons S1 0}
```

13

2. Sequential FP with single assignment

- We generate 10 elements
  - S1 is augmented every second
  - S2 is not displayed until after 10 seconds
  - S1 is created incrementally
    - Why?
    - Translate to kernel language!
  - S2 is created as a batch
    - Why?

```
% Prod uses unbound variables
fun {Prod L H}
   {Delay 1000} % Wait 1000 ms
   if L>H then nil
   else
      % L|S has unbound S
      L|{Prod L+1 H}
   end
end
```

```
declare S1 S2 in
{Browse S1}
{Browse S2}
S1={Prod 1 10}
S2={Cons S1 0}
```

14

3. Deterministic dataflow

- We execute both calls in their own threads
  - This is running deterministic dataflow (eager)
  - What is the difference with the previous version?
    - Both S1 and S2 are created incrementally
    - Each thread is a declarative agent

```
declare S1 S2 in
{Browse S1}
{Browse S2}
thread S1={Prod 1 10} end
thread S2={Cons S1 0} end
```

15

4. Lazy evaluation

```
fun lazy {Prod L H}
   {Delay 1000}
   if L>H then nil
   else L|{Prod L+1 H}
   end
end

fun lazy {Cons S Acc}
   case S of H|T then
      Acc+H|{Cons T Acc+H}
   [] nil then nil
   end
end
```

- We annotate both functions as "lazy"
- We execute it:

```
declare S1 S2 in
{Browse S1}
{Browse S2}
S1={Prod 1 10}
S2={Cons S1 0}
```

- What is going on?
  - Why is nothing computed?
  - How do we run this?
  - {Browse S2.2.1} needs the second element of S2, which will activate its computation and display it

16

Eager versus lazy streams

- One way to understand the difference between eager and lazy is to see which agent is driving the execution
- In an eager stream, it is the producer that determines when elements are sent
  - Termination is decided by the producer
- In a lazy stream, it is the consumer that determines when elements are sent
  - Termination is decided by the consumer

17

Infinite lists

18

Infinite lists

- With lazy evaluation we can compute with infinite loops
  - We can write programs with infinite lists
  - It works because the execution only computes needed elements
  - This is not possible in deterministic dataflow!
- An infinite list of integers starting with N:
  fun lazy {Ints N} N|{Ints N+1} **end**
- Calling {Ints 1} displays an unbound variable:
  L={Ints 1} {Browse L}
- We can force a computation by examining the list L:
  {Browse L.1}
  {Browse L.2.1}

19

Semantics of infinite lists

- We can see how infinite lists work by translating to kernel language:

```
proc {Ints N R}
    thread
        {WaitNeeded R} R=N|{Ints N+1}
    end
end
```

- When we need R by doing {Browse R.1}, this causes R to be bound to N|{Ints N+1}
  - This causes one element of R to be computed
  - The recursive call will immediately suspend again

20

Forcing a computation

- We can force the evaluation of N elements of a list by traversing the list:

```
proc {Touch L N}
   if N==0 then skip
   else {Touch L.2 N-1} end
end
```

- This strange procedure does nothing by itself, yet it forces the work to be done:

```
{Touch L 10}
{Touch L 20}
```

21

Hamming problem

22

Hamming problem

- Richard Hamming (1915-1998) was an engineer and mathematician who worked at Bell Labs and invented many useful things
  - Hamming codes, Hamming window, Hamming distance, etc.
  - *The Art of Doing Science and Engineering: Learning to Learn*, by Richard Hamming, 1997. This book is highly recommended!
- Today we will investigate the Hamming problem, a simple problem in number sequences
  - It is a dynamic problem where we do not know in advance how much needs to be computed → perfect for lazy evaluation!
  - We will use lazy evaluation to design a simple and efficient solution to this problem

23

Hamming problem

- Problem statement:
  - Given the set of numbers of the form $2^a\ 3^b\ 5^c$ with integers a,b,c ≥ 0
  - It is asked to compute these numbers in increasing order: 1 | 2 | 3 | …
- We do not know in advance how many numbers of this sequence will be needed
  - The program should let us compute them incrementally until we are satisfied
  - The program should be efficient in time and memory!

24

Algorithm idea

H = 1 | 2 | 3 | X |…

2H = 2 | **4** | 6 | …

3H = 3 | **6** | 9 | …

5H = **5** | 10 | 15 | …

$\Rightarrow$ X = min(4,6,5) = 4

- Idea: The next number X is 2 times, 3 times, or 5 times one of the previous numbers in the sequence
- We need to keep three sequences derived from H, namely 2H, 3H and 5H, and take the least number not yet used
- Numbers 2 and 3 are already taken
- Next number is either 4, 6, or 5
- We take the minimum of these three: the next number is 4

- We can program this with lazy lists

25

Hamming program operations

- The algorithm needs two operations
  - Multiply list elements by an integer
  - Merge two ordered lists
- L2={Times L1 N}
  - Each element of L2 is N times the element of L1
- L={Merge L1 L2}
  - Assume L1 and L2 are in increasing order
  - L contains elements of L1 and L2 in increasing order

26

Hamming program

```
fun lazy {Times S N}
    case S of H|T then
         N*H|{Times T N}
    end
end

fun lazy {Merge S1 S2}
    case S1|S2 of (H1|T1)|(H2|T2) then
         if H1<H2 then H1|{Merge T1 S2}
         elseif H1>H2 then H2|{Merge S1 T2}
         else /* H1==H2 */ H1|{Merge T1 T2}
         end
    end
end
```

- Main expression:

```
H=1|{Merge
       {Times H 2}
       {Merge {Times H 3}
              {Times H 5}}}
{Browse H}
```

27

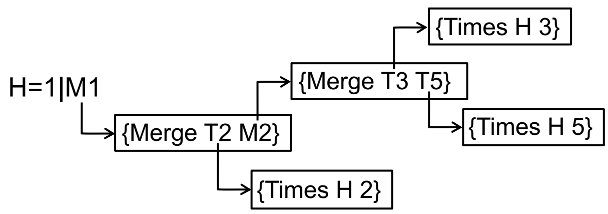
Lazy suspensions

28

Lazy suspensions

- We defined lazy evaluation using threads and WaitNeeded
  - This is correct but it does not show the execution
- Let us show the execution of a lazy program with a graphical approach
- We introduce the concept of lazy suspension:
  Executing: L2={Times L1 3}
  Creates a suspension: L2 → {Times L1 3}
  "A thread is suspended on L2 that contains the body of {Times L1 3}"

Execution of Hamming program



- Running the program creates five lazy suspensions
  - The lazy suspension {Merge T2 M2} waits on M1
  - Executing M1.1 activates the lazy suspension {Merge T2 M2}, which executes the body of {Merge T2 M2}, which then activates {Merge T3 T5} and {Times H 2}, and so forth!
    - All five lazy suspensions are activated and five new ones are created
    - At the end, M1.1 is bound to 2|M1' with the new variable M1'

First activation: {Merge T2 M2}

- Request the second element of H: {Browse M1.1}

- This activates {Merge T2 M2}:
  - The body is executed:
    case T2|M2 **of** (H1|T2') | (H2|M2') **then**
    …
    end
    Activate {Times H 2}
    Activate {Merge T3 T5}
  - The **case** needs the first elements of T2 and M2
    - This activates {Times H 2} and {Merge T3 T5}
    - The **case** waits patiently until T2 and M2 are bound

31

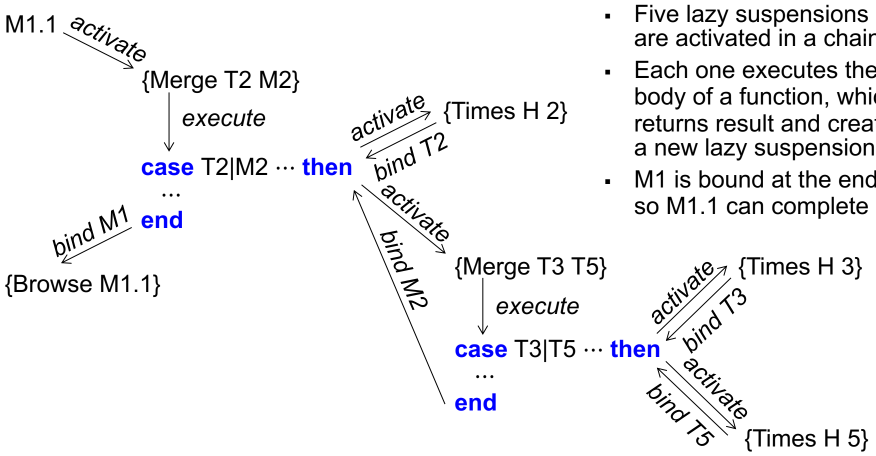
Next activations

- {Times H 2} and {Merge T3 T5} are activated
  - The body of {Times H 2} is executed
    - This binds T2=2|T2' and creates a new lazy suspension on T2': {Times M1 2}
  - The body of {Merge T3 T5} is executed
    - This activates {Times H 3} and {Times H 5}
    - After executing these two functions, this binds M2=3|M2' and creates a new lazy suspension on M2': {Merge T3' T5}
- Now the **case** in {Merge T2 M2}, which was waiting patiently, can be executed:
    - It returns 2|M1' with M1'={Merge T2' M2} and creates a new lazy suspension on M1'

32

Overall execution flow

The execution is sequential (follow the flow!)



- Doing M1.1 starts it all
- Five lazy suspensions are activated in a chain
- Each one executes the body of a function, which returns result and creates a new lazy suspension
- M1 is bound at the end so M1.1 can complete

33

Lazy deterministic dataflow

34

Five functional paradigms

- So far we have seen four paradigms of functional programming:
  - Sequential functional programming (no single assignment)
    - Traditional functional languages do this (Lisp, Scheme, ML, OCaml)
  - Sequential functional programming with single assignment
    - Allows data structures with “holes”, e.g., list functions are tail-recursive
    - This is the default way that Oz executes functions
  - Deterministic dataflow
    - Adds threads and dataflow synchronization
    - Allows concurrent programming with streams (multi-agent programming)
  - Lazy evaluation
    - Adds by-need synchronization (with WaitNeeded), where functions are executed only when their results are needed
    - Allows programming with infinite data structures
    - Lazy functional languages do this (Haskell, Miranda)
- There is a fifth paradigm:
  - Lazy deterministic dataflow
    - Adds both threads and lazy functions

35

Lazy deterministic dataflow

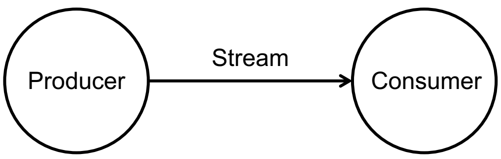
- Lazy deterministic dataflow is the most powerful declarative paradigm:
  - It has confluence and higher-order: the power of functional programming
  - It has concurrency: independent activities which can get out of step
  - It has lazy evaluation: by-need computations only done when needed
- What can we do with all this power?
  - We give one example of a program that can be written in lazy deterministic dataflow, but not in any weaker declarative paradigm
  - This program is the bounded buffer

36

Bounded buffer

37

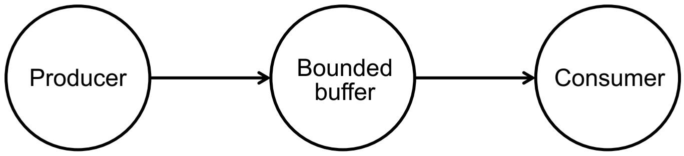
Bounded buffer (1)



- A producer-consumer pipeline has performance problems
  - Variations in producer and consumer speeds can cause the system to perform poorly
  - When a producer creates elements too quickly, the consumer cannot use the elements so the producer idles
  - When a consumer needs more elements, the producer may not be able to produce them so the consumer idles

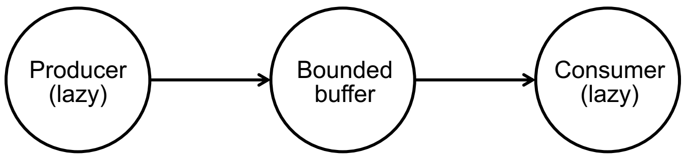
38

Bounded buffer (2)



- Inserting a bounded buffer can solve these problems
  - When the producer creates elements too quickly to be consumed, they are stored in the bounded buffer
  - When the consumer needs more elements than can be produced, they are taken from the bounded buffer
  - This improves performance by smoothing out fluctuations in producer and consumer speeds

Bounded buffer (3)



- A bounded buffer fits in between a lazy producer and a lazy consumer
- The code of the producer and consumer is unchanged
  - To the producer, the bounded buffer looks like a consumer
  - To the consumer, the bounded buffer looks like a producer
- The bounded buffer “consumes” elements even when the consumer does not ask for them, and “produces” elements even when the producer does not make them

Defining the bounded buffer

- Assume we have a producer-consumer pipeline:
  thread S={Producer …} **end**
  thread {Consumer S} **end**

- The bounded buffer is inserted in between:
  thread S1={Producer …} **end**
  thread {BoundedBuffer S1 S2 10} **end**
  thread {Consumer S2} **end**

- We define the bounded buffer step-by-step
  - We define the procedure {BoundedBuffer S1 S2 N} where S1 is the input stream, S2 is the output stream, and N is the buffer size
  - We build the procedure in four steps, to make it easier to understand

41

First step: pass elements

- The buffer outputs the same elements as it inputs:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1}
      case S1 of H1|T1 then H1|{Loop T1} end
   end
in
   S2={Loop S1}
end
```

42

Second step: startup

- The buffer asks for N elements on startup:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1}
      case S1 of H1|T1 then H1|{Loop T1} end
   end
   End
in
   End={List.drop S1 N} % Asking must not be lazy!
   S2={Loop S1}
end
```

- {List.drop L N} is a library function that removes the first N elements from a list L

43

Third step: staying full

- Whenever the consumer gets an element, the buffer asks for another element from the producer:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1 End}
      case S1 of H1|T1 then H1|{Loop T1 End.2} end
   end
   End
in
   End={List.drop S1 N}
   S2={Loop S1 End}
end
```

44

Fourth step: no blocking

- To avoid blocking the buffer's main loop, both asks must be done in their own threads:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1 End}
      case S1 of H1|T1 then
         H1|{Loop T1 thread End.2 end}
      end
   end
   End
in
   thread End={List.drop S1 N} end
   S2={Loop S1 End}
end
```

In declarative programming, threads are your friends! They are efficient. They can be added at will without adding bugs. They remove blocking and make the program more incremental.

All list functions, including List.drop, work correctly when used concurrently

Example execution

- We create a pipeline with producer, bounded buffer, and consumer:

```
declare S1 S2 S3 in
{Browse S1}
{Browse S2}
{Browse S3}
S1={Prod 1 10}
{BoundedBuffer S1 S2 3}
S3={Cons S2 0}
```

- Note that the producer immediately produces 3 elements, which are stored in the buffer
- When we consume one element, the buffer asks the producer for one element
  - The buffer tries to stay full
- The buffer is eager until it is full, and then it becomes lazy

Lazy quicksort

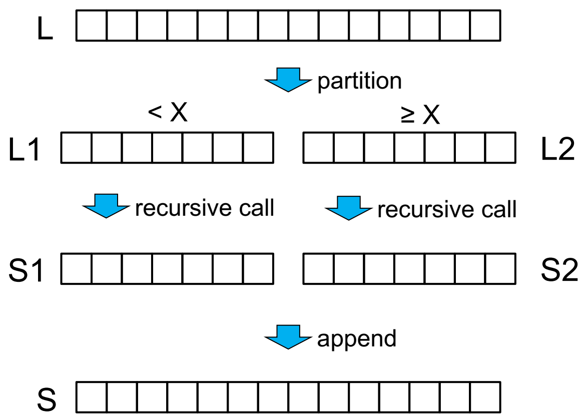
47

Lazy quicksort

- Lazy evaluation can make some algorithms incremental, which can enormously improve their efficiency
  - We show this with the quicksort algorithm
- Standard quicksort has an average time complexity of O(n log n) to sort n elements
- Lazy quicksort has a time complexity of O(n + k log k) to compute the k smallest elements out of n elements
  - This is a very good bound!
  - Furthermore, the value of k does not need to be known in advance. Elements can be computed incrementally until some condition is satisfied.
  - To see how clever this is, try inventing the algorithm from scratch!

48

Quicksort algorithm



- Pick a random element of L, the “pivot” X
- Partition into two sublists
- Recursively sort the sublists
- Append the results

49

Quicksort example (on board)

- L = [7 3 2 8 6 4 1 9]
- Pivot = 7 (first element of L)
- L1 = [3 2 6 4 1], L2 = [7 8 9]
- …
- S1 = [1 2 3 4 6], S2 = [7 8 9]
- S = [1 2 3 4 6 7 8 9]

50

Partition procedure

```
proc {Partition L X L1 L2}
   case L of H|T then
      if H<X then M1 in
         L1=H|M1 {Partition T X M1 L2}
      else /* H≥X */ M2 in
         L2=H|M2 {Partition T X L1 M2}
      end
   [] nil then L1=nil L2=nil
   end
end
```

51

Append and quicksort

```
fun {Append L1 L2}
   case L1 of H|T then H|{Append T L2}
   [] nil then L2 end
end
fun {Quicksort L}
   case L of X|M then L1 L2 S1 S2 in
      {Partition L X L1 L2}
      S1={Quicksort L1}
      S2={Quicksort L2}
      {Append S1 S2}
   [] nil then nil
   end
end
```

52

Example eager execution

- Let us try to run this:
  declare S in
  S={Quicksort [4 3 2 5 6 4 3 2]}
  {Browse S}
- What happens?
  - Something is wrong!
- How do we fix this?
  - A general rule when defining recursive functions!

53

Append and quicksort (fixed)

```
fun {Append L1 L2}
   case L1 of H|T then H|{Append T L2}
   [] nil then L2 end
end                        M is strictly smaller than L
fun {Quicksort L}
   case L of X|M then L1 L2 S1 S2 in
      {Partition M X L1 L2}
      S1={Quicksort L1} % L1 is strictly smaller than L
      S2={Quicksort L2} % L2 is strictly smaller than L
      {Append S1 X|S2}
   [] nil then nil
   end
end
```

54

Making quicksort lazy

- What has to be made lazy?
  - Quicksort function becomes LQuicksort
  - Append function becomes LAppend
- Partition is not lazy
  - Sorting cannot work unless we look at all the elements of L
  - Partition keeps the same eager definition
  - We create the complete sublists L1 and L2

55

Lazy append and quicksort

```
fun lazy {LAppend L1 L2}
   case L1 of H|T then H|{LAppend T L2}
   [] nil then L2 end
end
fun lazy {LQuicksort L}
   case L of X|M then L1 L2 S1 S2 in
      {Partition M X L1 L2}
      S1={LQuicksort L1}
      S2={LQuicksort L2}
      {LAppend S1 X|S2}
   [] nil then nil
   end
end
```

56

Example lazy executions

- Lazy append:
  declare S **in**
  S={LAppend [1 2 3] [4 5 6]}
  {Browse S}
  - What happens when asking for elements?

- Lazy quicksort:
  declare S **in**
  S={LQuicksort [4 3 2 5 6 4 3 2]}
  {Browse S}
  - What happens when asking for the first element?
  - How much computation is done? What is the time complexity?

57

Execution steps…

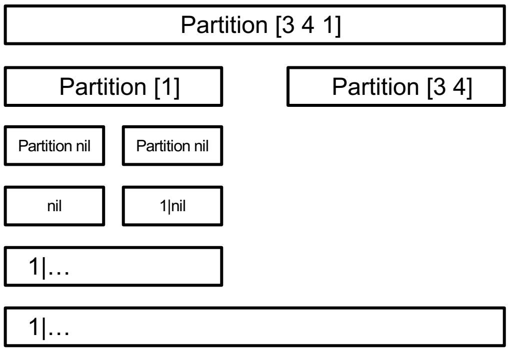
```
S={LQuicksort [2 3 4 1]} % Lazy suspension on S
{Browse S.1}
   % S is needed, so execute body of S={LQuicksort [2 3 4 1]}:
   {Partition [3 4 1] 2 L1 L2}
   S1={LQuicksort [1]} % Lazy suspension on S1
   S2={LQuicksort [3 4]} % Lazy suspension on S2
   S={LAppend S1 2|S2} % Lazy suspension on S
      % S still needed, so execute body of LAppend:
      case S1 of H|T then H|{LAppend T 2|S2}
      [] nil then 2|S2 end
         % S1 is needed, so execute body of S1={LQuicksort [1]}
         {Partition nil 1 nil nil}
         S1'={LQuicksort nil} % Lazy suspension on S1'
         S2'={LQuicksort nil} % Lazy suspension on S2'
         S1={LAppend S1' 1|S2'} % Lazy suspension on S1
            % S1 still needed, so execute body of LAppend:
            case S1' of H'|T' then H'|{LAppend T' 1|S2'}
            [] nil then 1|S2' end
               % S1' is needed, so execute body of S1'={LQuicksort nil}:
               case nil of X'|M' then (…)
               [] nil then nil end
               % Now we can do bindings:
               S1'=nil
         S1=1|S2'
   S=1|{LAppend nil 2|S2}
{Browse (1|…).1}
% Displays 1
```

- Follow carefully what is happening
  - When S is needed, it stays needed!
  - We focus on the lazy suspensions

- S → {LQuicksort [2 3 4 1]}
  S is needed, activates:
  - S1 → {LQuicksort [1]}
  - S2 → {LQuicksort [3 4]}
  - S → {LAppend S1 2|S2}
    S still needed, activates:
    S1 is needed, activates:
    - S1'→{LQuicksort nil}
    - S2'→{LQuicksort nil}
    - S1→{LAppend S1' 1|S2}
      S1 still needed, activates:
      S1' is needed, activates:
      - S1'=nil
    - S1=1|S2'
  - S=1|{LAppend nil 2|S2}

58

Complexity of lazy quicksort



n traversal

n/2 traversal
(n/4, n/8, continues to 1)

- To compute the smallest element, the number of operations is n + n/2 + n/4 + … + 1 = 2n, so the time complexity is O(n)
- To compute the k smallest elements, a full "mini quicksort" is done as soon as the partitioned list has at least k elements, so the extra time complexity is O(k log k)
- Total time complexity is O(n + k log k)

59

What is declarative programming?

60

Declarative programming

- We have seen five functional paradigms
  - Sequential functional programming
  - Sequential functional programming with single assignment
  - Deterministic dataflow (concurrent)
  - Lazy evaluation
  - Lazy deterministic dataflow (concurrent)
- We claim that they are all declarative
  - What does this mean, exactly?
  - Let us define it starting from the functional programming paradigm
- We show how to classify declarative paradigms according to their concepts and expressive power (Section 4.5.2 in the book)

61

Functional programming

- All functional programs can be encoded as λ expressions
- Church-Rosser theorem:

$$\begin{array}{ccccccc} e_a & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow \cdots \rightarrow & e_b \\ \downarrow & & & & & & \downarrow \\ e_2' & & & & & & \\ \downarrow & & & & & & \vdots \\ e_3' & & & & & & \\ \downarrow & & & & & & \downarrow \\ \vdots & & & & & & \vdots \\ e_c & \rightarrow & e_{c1} & \rightarrow & e_{c2} & \rightarrow \cdots \rightarrow & e_d \end{array}$$

- If e_a reduces to e_b (in 0 or more steps) and e_a reduces to e_c (in 0 or more steps), then there exists a term e_d such that e_b and e_c can reduce to e_d
- We say the λ calculus is confluent; it has the Church-Rosser property

62

Other functional paradigms?

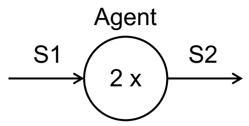
- We see that functional programs are confluent
  - The meaning is clear for the first paradigm, namely sequential functional programming
- But what does it mean for:
  - Concurrency? (threads and their scheduler)
  - Streams? (programs that never terminate!)
  - Single-assignment variables? (variables can be unbound!)
- We give a precise formal definition of “declarative programming” which covers these concepts
  - Confluence: this handles concurrency (why?)
  - Partial termination
  - Equivalent stores

63

I: Partial termination

64

Partial termination



- Assume we have a concurrent agent with an input stream S1 and an output stream S2
- It could execute as follows:
  - S1=1|_ S2=2|_
  - S1=1|2|_ S2=2|4|_
  - S1=1|2|3|_ S2=2|4|6|_
- How is this functional?
  - Problems: (1) the program never terminates and (2) the streams contain unbound variables
- With the right concepts, we can see this as functional execution:
  - If S1 does not change, then S2 reaches a final value
  - We call this “partial termination”
  - We say the program has reached a “resting point”
- What about the unbound variables?
  - See next section!

65

II: Equivalent stores

66

Single-assignment variables

- We claim that a functional program that uses single-assignment variables is still functional
  - Let's see how to make this precise
- Consider the following program:
  T_1: **thread** X=foo(Z W) **end**
  T_2: **thread** Y=foo(Z W) **end**
  T_3: **thread** X=Y **end**
  - Assume T_1 and T_2 execute before T_3, then we have the store: σ={*x*=foo(*z w*), *y*=foo(*z w*)}
  - Assume T_1 and T_3 execute before T_2, then we have the store: σ'={*x*=foo(*z w*), *y*=*x*}
- How can we express that stores σ and σ' are the same?
  - We first need to introduce some concepts from formal logic → Intro slides

67

A store is a logical formula

- Assume we have these two stores:
  σ={*x*=foo(*z w*), *y*=foo(*z w*)}
  σ'={*x*=foo(*z w*), *y*=*x*}
- The bindings of *x* and *y* are different for σ and σ' but the possible values of *x* and *y* are the same in both stores
  - Let's see how to make this intuition precise
- A store σ corresponds to a relationship between values
  - The store σ tells us that *x* is a record with label foo and arguments *z* and *w*, and that *y* is a record with label foo and arguments *z* and *w*
  - For all values of *x*, *y*, *z*, and *w*, there are two possibilities: either they can be part of a store σ or they cannot be part of a store σ
  - So the store σ is a logical formula, which can be true or false
  - We write σ as a logical formula: σ ≡ x=foo(z w) ∧ y=foo(z w)

68

Equivalent stores

- Now we can define when two stores are equivalent
  - Each store represents a logical formula that can be **true** or **false**
  - Two stores are equivalent when, no matter how we assign values to their symbols, they are either both **true** or both **false**
    - I.e., we cannot find values such that one store is **true** and the other is **false**
- We state this definition using the model concept
  - We introduce the notation $\alpha \vDash \beta$ which means "β is true in all models of α"
- Definition: Two stores σ and σ' are logically equivalent if
  - $\sigma \vDash \sigma'$ and $\sigma' \vDash \sigma$ $\quad$ σ' is **true** in all models of σ and σ is **true** in all models of σ'
- Another way to write this is:
  - $\vDash (\sigma \Leftrightarrow \sigma')$ $\quad$ $(\sigma \Leftrightarrow \sigma')$ is a tautology, i.e., it is **true** in all models

69

Introduction to first-order logic

70

First-order logic

- To define store equivalence, we introduce first-order logic
- Formal logic is:
  - A formal language: a syntactic definition of a set of formulas
  - A proof theory: a set of rules to deduce whether a formula is true or false, given a set of primitive formulas (axioms)
  - A model theory: mathematical objects in which the axioms are true
- First-order logic is:
  - A formal logic with variables, quantifiers, predicates, and connectors
    - Axiom: ∀x.∀z.(grandparent(x,z) ⇔ ∃y.parent(x,y) ∧ parent(y,z))
    - Model: any set of human beings with the parent relation
  - Some popular programming languages based on first-order logic are Prolog, constraint programming, and SQL

71

Example of first-order logic

- Formulas:
  syntactic expressions
  (with variables and quantifiers)

$\forall x.\ x<x+2$

$\exists x.\ (x^2-3x+2=0)$

$\forall x.\ (x^2-3x+2=0 \Rightarrow x=1 \lor x=2)$

$\forall x.\ \exists y.\ y=x^2$

- Models:
  integers $\mathbb{Z}$
  $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

  reals $\mathbb{R}$
  $\mathbb{R} = \{x \mid x \text{ has infinite decimal expansion}\}$

- All formulas on the left are true in $\mathbb{Z}$ and $\mathbb{R}$

72

Example of proof and model

- Given a set of formulas F
  - Given any formulas α, β ∈ F
- Proof theory
  - Given a set of proof rules
  - **α ⊢ β** : β can be deduced from α using the rules
  - $(\exists x.x^2=1) \vdash \exists x.(x=1 \vee x=-1)$
- Model theory
  - **α ⊨ β** : β is true in all models in which α is true
  - If $(\exists x.x^2=1)$ true in $\mathbb{R}$ then $\exists x.(x=1 \vee x=-1)$ true in $\mathbb{R}$

73

Model of a store σ

- We give a formal definition of a model of a store σ
- First step: An interpretation of a store σ (is true or false)
  - An interpretation of a store σ is an assignment to all symbols in σ
    - For all variables *x* in σ, assign a value **x** to *x*
    - For all record symbols *f* in σ, assign a function **f** to *f* that has the same number of arguments as the record symbol and that returns a value
  - Any interpretation of a store σ is either **true** or **false**
    - A binding $x=f(x_1 \dots x_n)$ is **true** if the value returned by $\mathbf{f}(\mathbf{x}_1 \dots \mathbf{x}_n)$ is equal to **x**; otherwise it is **false**
    - A store $\sigma = (x=f(x_1 \dots x_n) \wedge \dots \wedge z=f(z_1 \dots z_n))$ is **true** if all bindings are **true**, otherwise it is **false**
- Second step: A model of a store σ (is always true)
  - A model of σ is an interpretation in which σ is **true**

74

III: Definition of declarative programming

75

Definition of declarative programming

- Now we can define precisely what declarative programming means

A program is declarative if for all possible inputs:
- All executions for those inputs either:
  - do not terminate, or
  - all reach partial termination and give logically equivalent stores

- Remarks:
  - "All executions" means all possible choices of the scheduler
  - We say that a declarative program has "no observable nondeterminism"
  - All five functional paradigms are declarative

76

IV: Failure confinement

77

Fixing a buggy application

- Declarativeness is an extremely powerful property
  - How do we write applications to be as declarative as possible?
    - This is a major theme of the course! “All programs should be declarative except where they interact with the real world.”
  - How do we fix an application that becomes nondeclarative?
    - We can do failure confinement
- Nondeclarative behavior
  - We will see later in the course that applications that interact with the real world can be nondeclarative
    - That kind of nondeclarativeness is unavoidable but can be minimized
  - Right now, let us see what happens when an application has a bug that makes it nondeclarative

78

Bugs are unavoidable

- "It is a truth universally acknowledged, that a program of a certain size must have bugs"
  - With apologies to Jane Austen 😀
- Assume we have the following (simplified!) buggy program:
  thread X=1 **end**
  thread Y=2 **end**
  thread X=Y **end**
- This program will always raise an exception
  - Three stores are possible depending on the scheduler choices: $\sigma_1=\{x=1,y=2\}$, $\sigma_2=\{x=1,y=1\}$, $\sigma_3=\{x=2,y=2\}$
  - This is an observable nondeterminism, so it is nondeclarative
- We can fix this by doing failure confinement
  - We will hide the nondeterminism from the rest of the program
  - That way the program becomes declarative again

79

Failure confinement

- The program has three parts that can become inconsistent if there is a bug
  - We use exceptions to protect these parts

```
thread try X1=1 S1=ok catch _ then S1=error end end
thread try Y1=2 S2=ok catch _ then S2=error end end
thread try X1=Y1 S3=ok catch _ then S3=error end end
if S1==error orelse S2==error orelse S3==error then
   X=1  Y=1  /* default result when there is an error */
else
   X=X1  Y=Y1  /* correct result when there is no error */
end
```

80

Table of declarative paradigms

Declarative paradigms

	sequential with values	*sequential with values and dataflow variables*	*concurrent with values and dataflow variables*
eager execution (strictness)	strict functional programming (e.g., Scheme, ML) (1)&(2)&(3)	declarative model (e.g., Chapter 2, Prolog) (1), (2)&(3)	data–driven concurrent model (e.g., Section 4.1) (1), (2)&(3)
lazy execution	lazy functional programming (e.g., Haskell) (1)&(2), (3)	lazy FP with dataflow variables (1), (2), (3)	demand–driven concurrent model (e.g., Section 4.5.1) (1), (2), (3)

(1): Declare a variable in the store
(2): Specify the function to calculate the variable's value
(3): Evaluate the function and bind the variable

(1)&(2)&(3): Declaring, specifying, and evaluating all coincide
(1)&(2), (3): Declaring and specifying coincide; evaluating is done later
(1), (2)&(3): Declaring is done first; specifying and evaluating are done later and coincide
(1), (2), (3): Declaring, specifying, and evaluating are done separately

Conclusions

83

Conclusions

- Lazy evaluation
  - Functions are evaluated only if their results are needed
  - This extends dataflow (Wait & Bind) with the WaitNeeded operation
  - Programs can use infinite lists and be made more incremental
  - Lazy evaluation can be combined with concurrency
- Declarative programming
  - "An application should be declarative except for real-world interaction"
  - We define precisely what is declarative programming
    - We give a precise definition of declarative programming using the concepts of confluence, partial termination, and logical equivalence
  - Declarativeness is an observational concept: a program can behave declaratively even if it is written in a nondeclarative paradigm
- Next lecture: Advanced declarative algorithm design
  - Declarative algorithms can be as efficient as nondeclarative algorithms

84

LINFO1131
Concurrent programming concepts

Lecture 4:
Advanced declarative algorithm design

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Motivation
  - Writing efficient algorithms in declarative paradigms
- Concepts used
  - Single assignment and lazy evaluation
  - Amortized and worst-case
  - Ephemeral and persistent
- Summary of the algorithms
- Difference list
  - A list representation with efficient ephemeral operations
- Naïve queue
- Amortized constant-time ephemeral queue
- Worst-case constant-time ephemeral queue
  - A short step to logic programming
- Amortized constant-time persistent queue
- Worst-case constant-time persistent queue
- Conclusions

2

Motivation

- Writing declarative applications
  - We would like to write as much of the program as possible in a declarative paradigm
- Writing algorithms in a declarative paradigm
  - An important question: is it possible to make efficient algorithms in declarative paradigms?
    - Is mutable state really needed for efficiency? Not always!
  - Declarative algorithms can often be efficient
    - We give many techniques for doing this
    - We use the declarative paradigms we saw before

3

Recommended book

- Many of the examples in this lecture are taken from the book
  - "Purely Functional Data Structures" by Chris Okasaki

- This book shows how to use lazy evaluation to define many efficient declarative algorithms
  - The book uses the Standard ML language with an explicit lazy evaluation operation

4

Concepts used

5

Concepts used

- Today's lecture is based on the following concepts
- Language concepts
  - Single-assignment variables
  - Lazy evaluation

  } We saw these in the previous lecture
- Complexity concepts
  - Amortized upper bound
  - Worst-case upper bound
- Algorithm concepts
  - Ephemeral data structure
  - Persistent data structure

6

Amortized and worst-case

- Amortized complexity
  - If n operations have a combined complexity of O(f(n)) then we say each operation has an amortized complexity of O(f(n)/n)
  - This is important when individual operations are sometimes expensive but operations are cheap on average
  - For example, individual operations on a queue can have complexity O(n), but with n operations it is possible for individual operations to have amortized complexity of O(1)
- Worst-case complexity
  - This is the big-O notation that you have seen before
- A good worst-case upper bound is best, but if this is not possible, an amortized bound may be good enough

7

Ephemeral and persistent

- An ephemeral data structure can have only one version exist at the same time
  - Given queue Q1, then doing Q2={Insert Q1 a} creates a new queue Q2 and Q1 can no longer be used
  - Stateful data structures (like in Java) are always ephemeral: when you change the attributes of an object, the old values are forgotten
- A persistent data structure can have many versions exist at the same time
  - Given queue Q1, then doing Q2={Insert Q1 a} creates a new queue Q2 and Q1 can still be used. Doing Q3={Insert Q1 b} will create another version, and all versions Q1, Q2, Q3 are still usable
  - What are persistent data structures good for?

8

The use of persistence

- Persistent data structures are used for collaborative work
  - Two people edit the same text and create their own versions
  - Software repositories like github support multiple versions
  - Some databases support multiple versions
- In systems with mutable state (like Java), multiple versions can be simulated by doing explicit “copy” operations, but this is clumsy
  - In our declarative code, the versions are handled automatically
- “Merge” operation
  - Persistent data structures often have a “merge” operation that allows to combine two versions
    - The merge takes care of possible conflicts between versions

9

Summary of the queues

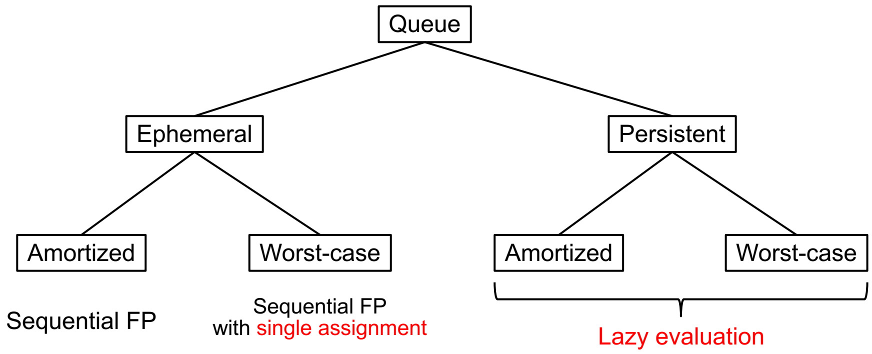
10

Summary of the queues

- We show five ways to implement a queue abstraction
  - We first give a naïve algorithm; the others have better properties!
  - We use three declarative paradigms for these implementations
- Sequential functional programming
  - Naïve queue, insert is O(1) and delete is O(n)
  - Amortized ephemeral queue, insert and delete O(1)
- Sequential functional programming with single assignment
  - Worst-case ephemeral queue, insert and delete O(1)
- Lazy evaluation
  - Amortized persistent queue, insert and delete O(1)
  - Worst-case persistent queue, insert and delete O(1)

11

Summary of the queues



12

Difference list

13

Difference list

- A difference list is a representation of a list as a pair of two lists (S,E) such that E is a suffix of S
- The difference list (S,E) represents the list S–E where we take S and remove the suffix E
  - The list 1|2|3|nil can be represented as a difference list:
    - As the pair (1|2|3|X,X) where X is an unbound variable
    - As the pair (1|2|3|4|Y,4|Y) where Y is an unbound variable
    - As the pair (1|2|3|4|5|nil,4|5|nil)
    - And so forth, there are an infinite number of possibilities
- What is the advantage of this representation?
  - A difference list has efficient ephemeral operations!

14

Adding an element

- Add an element to the start or the end in constant time
  - Given the difference list (S,E) where E is an unbound variable
  - Adding X to the start gives (S1,E) with the binding S1=X|S
  - Adding X to the end gives (S,E1) with the binding E=X|E1
- For example, take the difference list (1|2|3|X,X)
  - Adding 4 to the start gives (4|1|2|3|X,X)
  - Adding 4 to the end gives (1|2|3|4|Y,Y)
    - Bind X=4|Y, then take the first element of the original difference list together with Y, which gives (1|2|3|4|Y,Y)
- For a standard list, like 1|2|3|nil, it is not possible to add an element at the end in constant time!

15

Constant-time append

- We append two difference lists in constant time
  - Given (S1,E1) and (S2,E2), then the append is given by (S1,E2) with the binding E1=S2
  - For example, we append (a|b|X,X) and (1|2|Y,Y) by binding X=1|2|Y and taking the start of the first with the end of the second (a|b|1|2|Y,Y)
- This can only be done once (it is ephemeral)!
  - In many cases, that is all we need
  - Let us show an example how it can be used

16

Naïve flatten

- Append function

```
fun {Append L1 L2}
   case L1 of nil then L2
   [] H|T then
      H|{Append T L2}
   end
end
```

- Append is a tail-recursive function whose execution time is proportional to |L1|, i.e., the length of L1

- Naïve flatten function

```
fun {Flatten Xs}
   case Xs of nil then nil
   [] X|Xr andthen
     {IsList X} then
       {Append {Flatten X}
          {Flatten Xr}}
   [] X|Xr then
       X|{Flatten Xr}
   end
end
```

- {Flatten [[1 2] [3 4] 5]} gives [1 2 3 4 5]

17

Flatten with a difference list

- We replace the list result by a difference list
  - We add two arguments S and E
  - We define it as a procedure to show the arguments

```
proc {DFlatten Xs S E}
   case Xs of nil then
      S=E
   [] X|Xr andthen {IsList X} then M in
      {DFlatten X S M}
      {DFlatten Xr M E}
   [] X|Xr then M in
      S=X|M
      {DFlatten Xr M E}
   end
end
```

This is much more efficient than the naïve Flatten!

(S,E) is the append of (S,M) and (M,E)

18

Difference lists versus linked lists

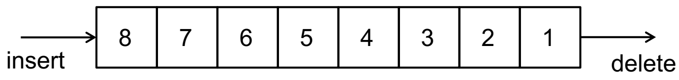
- These two data structures seem quite similar
  - Difference list (functional)
  - Linked list (stateful, for example in Java)
- What's the difference?
  - Both allow efficiently building chains of elements
  - The difference is that difference lists cannot be broken: they have functional semantics
  - Linked lists can always be broken: the chains can always be modified by assignment

19

Naïve queue

20

Queue abstraction



- A queue is a sequence that allows insert on one end and delete on the other end
  - Insert 1, 2, …, 8
  - First-in first-out (FIFO): delete 1, 2, …, 8
- We first give a naïve queue implementation in sequential functional programming

21

Naïve queue

- We define a simple queue abstraction
  - Not a true abstraction since the representation is not protected
  - What is its complexity?
- Helper function:

```
fun {ButLast L X}
   case L
   of [Y] then X=Y nil
   [] Y|L1 then
      Y|{ButLast L1 X}
   end
end
```

- Queue operations:

```
fun {NewQueue} nil end
fun {Insert Q X} X|Q end
fun {Delete Q X}
   {ButLast Q X}
end
```

- Example execution:

```
declare Q Q1 1 X1 in
Q={Insert {Insert {Insert
      {NewQueue} 1} 2} 3}
{Browse Q}
Q1={Delete Q X1}
{Browse X1}
```

22

Amortized constant-time ephemeral queue

23

Amortized ephemeral queue

- We define an amortized constant-time ephemeral queue in functional programming
- The queue is represented as a tuple q(F R)
    - Content is {Append F {Reverse R}}
    - Insert is done by updating R
    - Delete is done by updating F
    - If F is empty, then {Reverse R} is done to move elements from R to F

```
fun {NewQueue} q(nil nil) end
fun {Check Q}
   case Q of q(nil R) then
      q({Reverse R} nil) else Q end
end
fun {Insert Q X}
   case Q of q(F R) then
      {Check q(F X|R)} end
end
fun {Delete Q X}
   case Q of q(F R) then F1 in
      F=X|F1 {Check q(F1 R)} end
end
```

24

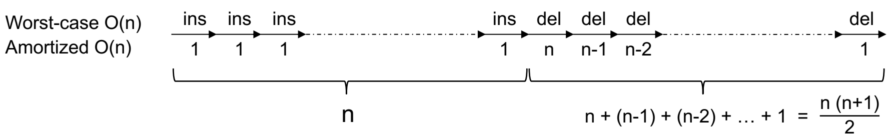
Discussion

- Example execution: (exactly as before!)
  Q={Insert {Insert {Insert {NewQueue} 1} 2} 3}
  {Browse Q}
  Q1={Delete Q X1}
  {Browse X1}

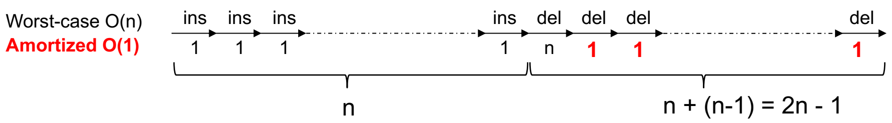
- Questions
  - If this queue is used in a persistent manner (with multiple versions) then the results will be correct. Why?
  - However, when used in persistent manner, this queue is no longer amortized constant-time. Why?
    - Hint: find a sequence of n operations that has worse complexity than O(n)

Comparing naïve queue with amortized queue

Naïve queue (single list L)



Amortized constant-time ephemeral queue (two lists F and R)



Worst-case constant-time ephemeral queue

27

Worst-case ephemeral queue

- It is not possible to write a queue with constant-time insert and delete in standard functional programming
  - Amortized constant-time is the best we can do
  - But adding single assignment makes it possible!
- The queue is represented as the tuple q(S E) where (S,E) is a difference list with the content
- Both insert and delete are always constant-time!

```
fun {NewQueue} X in q(X X) end
fun {Insert Q X}
   case Q of q(S E) then E1 in
      E=X|E1 q(S  E1)
   end
end
fun {Delete Q X}
   case Q of q(S E) then S1 in
      S=X|S1 q(S1 E)
   end
end
```

28

Knowing how many elements

- The previous definition does not let us know when the queue is empty
- To know the number of elements, we add the queue size to the representation
  - The queue is represented as q(N S E) where N is the number of elements and (S,E) is the same as before
  - Test if empty:
    fun {IsEmpty Q} Q.1==0 **end**
- Both insert and delete are still constant-time!

```
fun {NewQueue} X in q(0 X X) end
fun {Insert Q X}
   case Q of q(N S E) then E1 in
      E=X|E1 q(N+1 S  E1)
   end
end
fun {Delete Q X}
   case Q of q(N S E) then S1 in
      S=X|S1 q(N-1 S1 E)
   end
end
```

29

A short step to logic programming

30

Doing delete before insert

- Try the following execution:
  declare Q1 Q2 Q3 X **in**
  Q1={NewQueue}
  Q2={Delete Q1 X} % Delete from an empty queue
  {Browse X}
  Q3={Insert Q2 foo} % Insert an element

- This first displays an unbound variable X
  - When foo is inserted, the display is updated to foo
  - The delete creates an empty slot that is filled later by insert
  - How can this work?

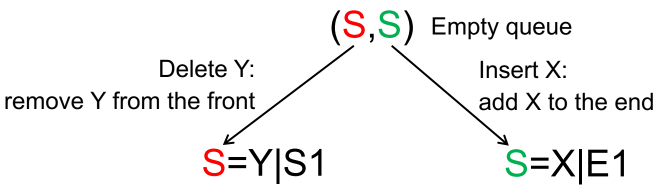
31

Special power of this queue

- This queue definition has a special power that follows from the logical equivalence property of stores
  - The queue can have a negative number of elements!
  - An element can be deleted before it is inserted
- The queue definition guarantees that deleted elements are equal to inserted elements
  - It is because binding done in any order gives the same results
    - Binding is a symmetric operation; the general binding operation is called unification and it follows from the logical equivalence of stores
  - We can delete an unbound variable first and insert a value later
- We are doing more than just functional programming
  - We are doing logic programming, similar to what Prolog does

32

Delete before insert explained



- We start with an empty queue (S,S)
- Let's do delete Y and insert X on the empty queue
  - This gives two bindings, S=Y|S1 and S=X|E1
- What happens in the store?
  - We have the logical formula ($s=y|s_1 \wedge s=x|e_1$)
  - Simplifying shows us that $y=x$ and $s_1=e_1$

33

Shoutout to Prolog

- The queue is doing logic programming
  - Both insert and delete do unification with the same variable
  - Because of logical equivalence, this imposes logical equality
  - Remember the last lecture's introduction to first-order logic!
- Logic programming is another declarative paradigm
  - Logic programming is more general than functional programming
  - Data structures are truths: they can have unbound variables and binding is bidirectional (both inputs and outputs can be bound)
  - Computation is deduction: a running program deduces new truths
    - A Prolog program is actually a theorem prover
- If you are curious, check out Prolog and constraint programming
  - See programming paradigms and constraint programming courses

34

Amortized constant-time persistent queue

35

Making it persistent

- Persistence is a strong property that is hard to get
  - As your program updates its data structures, many versions are created that exist simultaneously
  - Stateful programming, like in Java and Python, is ephemeral by default. All algorithms using mutable state are ephemeral by default!
    - Stateful algorithms can be made persistent by making explicit copies, but this is hard because it is managed by the programmer
- Declarative algorithms can be made persistent by using lazy evaluation
  - This is another amazing property of lazy evaluation

36

Helper function: lazy append

- We define a lazy append like we did before with quicksort
  ```
  fun lazy {LAppend Xs Ys}
     case Xs of X|Xr then X|{LAppend Xr Ys}
     [] nil then Ys end
  end
  ```
- Example execution:
  ```
  declare L in
  L={LAppend [1 2 3] [4 5 6]}
  {Browse L}
  ```
- Run this and ask for elements of L, to understand it!
  - It gives 1, 2, 3, and then [4 5 6] all at once

37

Persistent algorithm idea

- We define the queue again, but with yet another representation
- We use a tuple q(LenF F LenR R) where LenF and LenR are integers giving the length of F and R
  - As before, we move elements from R to F when F becomes empty
  - But now we do the move with a lazy suspension
- How we get amortized constant-time
  - The move does a {Reverse R} which cannot be made incremental
  - To make it amortized, we pay for the lazy suspension in advance
  - We use the “banker’s method”: we do n operations in advance before creating the lazy suspension
  - It is like saving money: save bit by bit and buy when you have enough

38

Persistent algorithm code

```
fun {NewQueue} q(0 nil 0 nil) end
fun {Check Q}
   case Q of q(LenF F LenR R) then
      if LenF<LenR then
         q(LenF+LenR {LAppend F {fun lazy {$} {Reverse R} end}} 0 nil)
      else Q end
   end
end
fun {Insert Q X}
   case Q of q(LenF F LenR R) then {Check q(LenF F LenR+1 X|R)} end
end
fun {Delete Q X}
   case Q of q(LenF F LenR R) then F1 in F=X|F1 {Check q(LenF-1 F1 LenR R)} end
end
```

Move R to F (lazily)

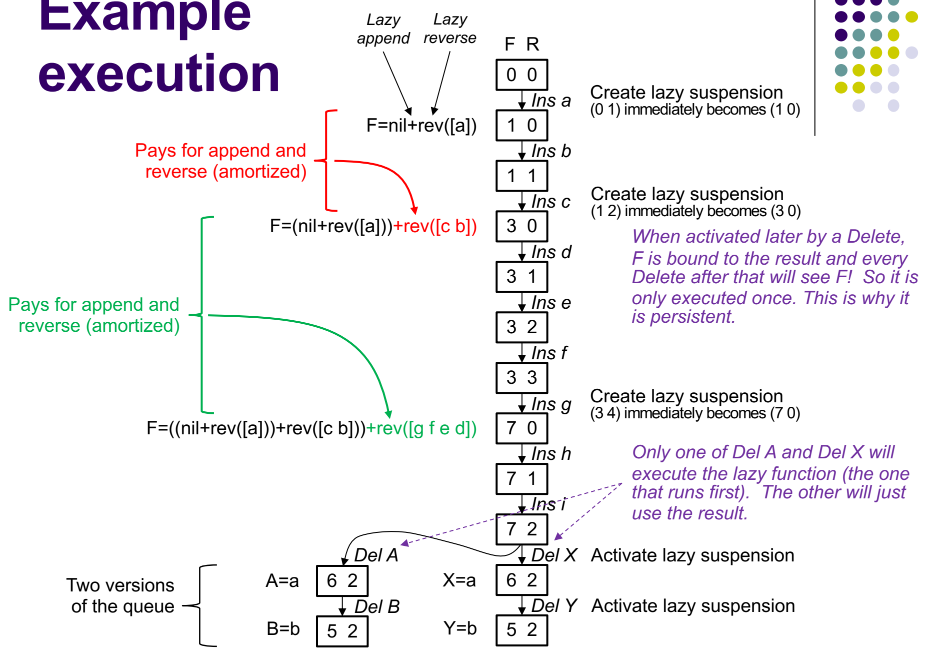
Increase R

Decrease F

How it works (example on next slide)

- The trick is to make sure the algorithm creates a lazy suspension at the right time
  - It has to be done when the {Reverse R} is paid for
    - Banker's method: save operations in the bank!
  - Assume we are inserting elements
    - This causes R to increase
    - When R is larger than F, create the lazy suspension
    - Activating the lazy suspension makes F bigger and R empty
  - Assume we are deleting elements
    - We activate a lazy suspension to get an element, and this triggers the {Reverse R}, but it's ok since it's paid for

Example execution



41

Why {Reverse R} is monolithic

- The {Reverse R} function cannot be made incremental by lazy evaluation
  - We say that it is monolithic
- It is because we cannot know the first element of the reversed list without traversing the whole list
  - Any function where we need to see the whole data structure in order to create a single output cannot be made incremental by lazy evaluation (another example: Partition in lazy quicksort)
- We show the code…
  - If you try to execute Reverse lazily you will see why this happens: the recursive calls of Reverse don't create any results until the recursion stops at the end

42

Execution of lazy reverse

- Reverse function:
```
fun lazy {Reverse L A}
   case L of X|L2 then
      {Reverse L2 X|A}
   [] nil then A
   end
end
```
- Traverse list L and build reverse in accumulator A

- Sample call:
  R={Reverse [1 2 3] nil}
- What happens when we ask for the first element?
  {Browse R.1}
- R is needed so the lazy suspension is activated and executes the body. This calls:
  {Reverse [2 3] 1|A}
- This creates another lazy suspension that is immediately activated because it is needed!
  - To get the first element, we keep traversing L to the end

43

Worst-case constant-time persistent queue

44

Achieving worst-case constant-time

- The reason why the previous example was amortized constant-time was because of the {Reverse R} call
  - Reverse is monolithic: it is executed all at once
- To fix this, we need to execute the reverse step by step
  - The old code is {LAppend F {**fun** lazy {$} {Reverse R} **end**}}
    - This code does both LAppend and Reverse
  - The trick is to merge them into a new function AppRev
    - Each time LAppend does one iteration, we do one step of Reverse
    - The execution of Reverse is “spread out” over n operations
- We show how to merge Append and Reverse

45

“Spreading out” the Reverse

- Old code: {LAppend F {**fun** lazy {$} {Reverse R} **end**}}
  - This code will first get elements lazily from F
  - When F is completely used up, then it executes {Reverse R}
  - It calculates all elements of {Reverse R} in one operation
- New code: {LAppRev F R nil}
  - The function LAppRev is like LAppend, but whenever it does one iteration of Append, it also does one iteration of Reverse
  - When the LAppRev is done (because F is completely used up), then the Reverse is completely executed!

46

Defining LAppRev

- We explain how we combine Append and Reverse
- Here is the code for Append and Reverse:

```
fun lazy {LAppend F B}
   case F of X|F2 then
      X|{LAppend F2 B}
   [] nil then B end
end
fun {Reverse R B}
   case R of Y|R2 then
      {Reverse R2 Y|B}
   [] nil then B end
end
```

- Here is the code for LAppRev that does both Append and Reverse:

```
fun lazy {LAppRev F R B}
   case pair(F R)
   of pair(X|F2 Y|R2) then
      X|{LAppRev F2 R2 Y|B}
   [] pair(nil [Y]) then Y|B
   end
end
```

- Notes:
  - **Green** arguments come from LAppend, **red** ones from Reverse
  - When F is empty, then R has one element left (due to F<R condition: R has grown bigger than F)

47

Persistent algorithm code (new version)

New code replaces old code

```
fun {NewQueue} q(0 nil 0 nil) end
fun {Check Q}
   case Q of q(LenF F LenR R) then
      if LenF<LenR then
         q(LenF+LenR {LAppRev F R nil} 0 nil)
      else Q end
   end
end
fun {Insert Q X}
   case Q of q(LenF F LenR R) then {Check q(LenF F LenR+1 X|R)} end
end
fun {Delete Q X}
   case Q of q(LenF F LenR R) then F1 in F=X|F1 {Check q(LenF-1 F1 LenR R)} end
end
```

Move R to F (lazily)

Increase R

Decrease F

48

Conclusions

49

Conclusions

- We define important algorithm concepts
  - Amortized complexity: single operations may be expensive but on average they are efficient
  - Persistence: multiple versions of data structures can be used
- We write efficient algorithms in declarative paradigms
  - We take a simple algorithm, a queue, and show four ways how it can be implemented efficiently
  - We use both lazy evaluation and single assignment
- As a bonus, we make a step toward logic programming
  - Because of logical equivalence of stores, variable binding is actually a symmetric operation called unification
  - Logic programming is the most powerful form of declarative programming – check out Prolog and constraint programming

50

Take-away intuitions

- Concepts for efficient declarative algorithms
  - Single assignment for fast ephemeral algorithms
  - Lazy evaluation for fast persistent algorithms

- Why it works
  - Single assignment is a weak form of mutable state that is still declarative but is strong enough for ephemeral algorithms because they only do assignment once
  - Lazy evaluation lets expensive operations be done in advance (which improves behavior for multiple versions) and be decomposed into small steps (which improves behavior for worst-case)

LINFO1131
Concurrent programming concepts

Lecture 5:
Limitations of declarative programming

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Limitations of declarative programming
  - Declarative paradigms are based on lambda calculus: they are confluent but they do not interact with the real world during their execution
  - We explain how to extend declarative paradigms to interact with the real world, by adding imperative concepts such as mutable state or communication channels
- Cells
  - A form of mutable state that allows to overcome the limitations of declarative programming
  - This leads to shared-state concurrency
- Ports
  - A communication channel that allows to overcome the limitations of declarative programming
  - This leads to message-passing concurrency (multi-agent programming)

2

Limitations of declarative programming

3

Beyond declarative programming?

- Up to now we have seen only declarative paradigms
  - Sequential functional programming
  - Deterministic dataflow and lazy deterministic dataflow
  - Efficient declarative algorithms
  - These are powerful and useful paradigms!
- Ideally, your program should be completely declarative!
  - Correctness, testing, and maintenance are much simplified!
  - But unfortunately this is impossible
  - Why is it impossible? Let us see by looking at lambda calculus!
  - Luckily most programs only need a few nondeclarative bits, so most of the program can still be declarative

4

Declarative execution = lambda execution

- Declarative execution is equivalent to lambda execution:

$$e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ \dots \rightarrow e_{n-1} \rightarrow e_n$$

- Execution starts with initial expression e_0 and reduces it in steps, ending with final expression e_n
  - Lambda calculus is Turing complete, it can do all computations
  - The power is a consequence of the Church-Rosser theorem (confluence): final result e_n is independent of the reduction order
- How does this execution interact with the real world?
  - In declarative programming, it does not! All information is already in the initial expression e_0, nothing is added later.

5

Interacting with the real world is not declarative

- Practical programs take time to execute
  - Each reduction step $e_{i-1} \rightarrow e_i$ takes time because execution happens on a computer, a physical artifact in the real world
- Some reduction steps interact with the real world
  - They accept inputs (like s_0) or they generate outputs (like s_1)

$$\begin{array}{ccccccccccccc} & & & & s_0 & & & & s_1 & & & & \\ & & & & \downarrow & & & & \uparrow & & & & \\ e_0 & \rightarrow & e_1 & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow & e_4 & \rightarrow \dots \rightarrow & e_{n-1} & \rightarrow & e_n \end{array}$$

- This is not a lambda execution any more!
  - Because input s_0 affects the value of e_2, and because output s_1 comes from e_4 so it is not a lambda final expression

6

Lambda calculus: confluent but no real-world interaction

Confluent reduction of an initial expression to a final result

This has very strong mathematical properties that we can use

- For reasoning, debugging, testing, optimization, and maintenance
- For concurrency, parallelism, and distribution
- There is no efficiency penalty compared to other paradigms!

But it can't interact with the real world! Let's see why:

- During the execution, we would like to accept inputs coming from the real world and outputs going back to it
- Declarative programming can't interact with the real world because its execution is a step-by-step reduction of an initial expression to a final result. Reduction steps take time, and the inputs will arrive during this time. The reduction can't use them unless we could put them in the initial expression. But we can't do this, because the inputs are not known in advance.

Imperative programming

- To interact with the real world, we need to add something to the declarative paradigms
  - A way to receive inputs and send outputs during execution
  - This is usually called imperative programming
- This lets us interact with the real world, but we also have to give up the goodness of declarative programming
- Can we have our cake and eat it too? Both the good properties of declarative programming and interaction with the real world?
  - No we can't! So what can we do…?

The right way to design programs

- Write most of the program in a declarative paradigm
  - And add small pieces of imperative programming only in those places that interact with the real world
  - Usually there are only a very few such places, so we keep most of the advantages of declarative programming
- We can use this to improve existing systems too…
  - Existing systems are often not designed like this! They do too much imperative programming. Older systems like Java are especially bad.
  - This gives us a measure to judge how well existing systems are designed (and a way to improve them: make them more declarative)

9

9

Kinds of interactions

- There are many ways that a program can interact with the real world
- Here are three typical possibilities:
  - Hardware clock: Input s_k gives the clock time
  - Mutable state: output s_a writes to a register, later input s_b (with b>a) reads from the register
  - Communication channel: output s_b sends to a channel, later input s_c receives from the channel
- Executions give different results depending on the exact timing and order of the reductions

10

Hardware clock

- Here is an example of a hardware clock:

$$\begin{array}{ccccccccccccc} & & t_0 & & & & t_1 & & & & t_2 & & \\ & & \downarrow\text{read} & & & & \downarrow\text{read} & & & & \downarrow\text{read} & & \\ e_0 & \to & e_1 & \to & e_2 & \to & e_3 & \to & e_4 & \to & e_5 & \to \dots \to & e_{n-1} \to e_n \end{array}$$

- A read returns the current time from the clock
  - Reduction $e_0 \to e_1$ reads time t_0
  - Reduction $e_2 \to e_3$ reads time t_1
  - Reduction $e_4 \to e_5$ reads time t_2
- Exact time values depend on reduction timing and order
  - This is not lambda reduction, since for a lambda reduction the result is independent of reduction timing and order (confluence)

11

Mutable state

- Here is an example of a mutable variable:

$$\begin{array}{ccccccccccccc} & & x & & & & y & & & & y & & \\ & & \uparrow\text{write} & & & & \uparrow\text{write} & & & & \downarrow\text{read} & & \\ e_0 & \to & e_1 & \to & e_2 & \to & e_3 & \to & e_4 & \to & e_5 & \to \dots \to & e_{n-1} \to e_n \end{array}$$

- A read returns the value of the most recent write
  - Reduction $e_0 \to e_1$ writes x in the register
  - Reduction $e_2 \to e_3$ writes y in the register
  - Reduction $e_4 \to e_5$ reads y from the register (not x!)
- Result of reads depends on reduction order
  - This is not lambda reduction, since for a lambda reduction the result is independent of the reduction order (confluence)

12

Communication channel

- Here is an example of a FIFO channel:

$$
\begin{array}{ccccccccccccc}
 & & x & & y & & x & & y & & & & \\
 & & \uparrow\text{send} & & \uparrow\text{send} & & \downarrow\text{receive} & & \downarrow\text{receive} & & & & \\
e_0 & \rightarrow & e_1 & \rightarrow e_2 \rightarrow & e_3 & \rightarrow e_4 \rightarrow & e_5 & \rightarrow & e_6 & \rightarrow \ldots \rightarrow & e_{n-1} & \rightarrow & e_n
\end{array}
$$

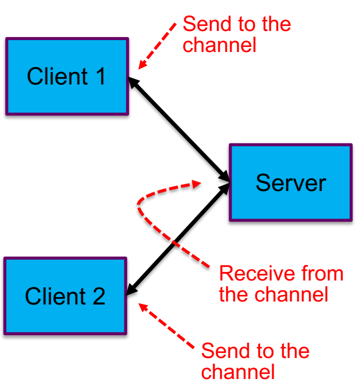
- The write happens before the read in the reduction order
  - Reduction $e_0 \rightarrow e_1$ sends x on the channel
  - Reduction $e_2 \rightarrow e_3$ sends y on the channel
  - Reduction $e_4 \rightarrow e_5$ receives x from the channel
  - Reduction $e_5 \rightarrow e_6$ receives y from the channel

Nondeterministic!

- Order of received values depends on reduction order
  - Sending of x and y might be reversed if they are concurrent!

13

Client/server application



- Let's use the communication channel to build a client/server application
  - To satisfy client liveness, the server must accept each incoming client request in a reasonable time that depends only on the travel time from the client to the server
- However, the order of the requests cannot be determined in advance because it depends on precise client timing (different timings give different reduction orders)
  - This means that the communication channel is nondeterministic
- The whole client/server application is therefore nondeterministic, even if all the other code is purely declarative

14

14

The two most important nondeclarative operations

- Two most important nondeclarative operations are mutable state and communication channels
  - In the course we will show how to use both of them
- Mutable state: called cells
  - Leads to shared-state concurrency (Java)
  - Locks, monitors, transactions
- Communication channels: called ports
  - Leads to message-passing concurrency (Erlang)
  - Multi-agent programming

15

Two definitions of declarative programming

16

Two definitions

- You will notice that we have made two definitions of declarative programming
  - "A program is declarative if for all possible inputs, all executions either do not terminate or they terminate and give logically equivalent results" (lecture 3)
    - This is an observational definition: we observe a program from the outside, we don't care how the program is implemented
  - "A program is declarative if it is equivalent to an execution of a program in lambda calculus" (lecture 5)
    - This is a structural definition: it is based on how the program is implemented

17

Comparing the definitions

- Observational is strictly more general than structural
  - All lambda executions are declarative when observed from the outside
  - An observational declarative program can be implemented using mutable state, as long as the state has no observable effect (it is hidden)
- The observational definition is best for designing programs
  - It captures the idea that the program must be deterministic even if it is concurrent, just as with lambda calculus (Church-Rosser theorem)
  - We can use mutable state in the implementation, as long as it is hidden
    - This is important because mutable state is a fundamental part of today's processors, so the low-level parts of the implementation must use it!
- The structural definition gives the theoretical basis
  - It shows that declarative programming is possible and practical
    - The Church-Rosser theorem is an important and nonobvious result!

18

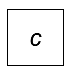
Cells

19

A cell

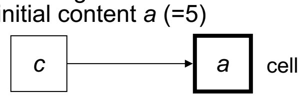
- To overcome the limitations of declarative programming, we add cells (mutable variables) to the language
- A cell is a box with identity and content
  - The identity is a constant (the “name” or “address” of the cell)
  - The content is a variable (in the single-assignment store)
- The content can be replaced by another variable

```
A=5
B=6
C={NewCell A}  % Create a cell
{Browse @C}    % Display content
C:=B           % Change content
{Browse @C}    % Display content
```

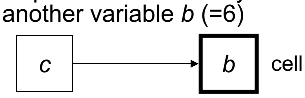


An unbound variable

Creating a cell with initial content *a* (=5)



Replace the content by another variable *b* (=6)



20

Adding cells to the kernel language

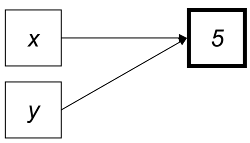
- We add cells and their operations
  - Cells have three operations
- C={NewCell A}
  - Create a new cell with initial content A
  - Bind C to the cell's identity
- C:=B
  - Check that C is bound to a cell's identity
  - Replace the cell's content by B
- Z=@C
  - Check that C is bound to a cell's identity
  - Bind Z to the cell's content

{Exchange C Z B}

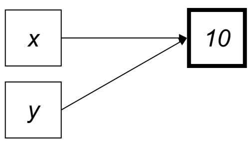
Cell examples (1)

- X={NewCell 0}

- X:=5
- Y=X

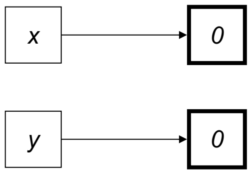


- Y:=10
- @X==10 % true
- X==Y % true



Cell examples (2)

- X={NewCell 0}
- Y={NewCell 0}



- X==Y % false
- Because X and Y refer to different cells, with different identities

- @X==@Y % true
- Because the contents of X and Y are the same value

23

Cell semantics (1)

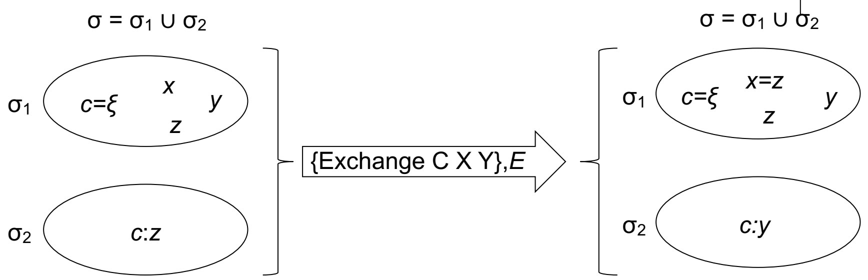
- Assume single-assignment store σ_1 with variables
- Assume cell store σ_2 that contains pairs of variables
- Full store $\sigma = \sigma_1 \cup \sigma_2$

environment

- C={NewCell X}, {C→*c*, X→*x*}
  - Assume variables *c*, $x \in \sigma_1$ (*c* is unbound)
  - Create fresh name ξ, bind $c=\xi$, add pair $c{:}x$ to σ_2
- {Exchange C X Y}, {C→*c*, X→*x*, Y→*y*}
  - Assume $c=\xi$, variables x, y, $z \in \sigma_1$, $c{:}z \in \sigma_2$
  - Bind $x=z$ (get old value), update pair to $c{:}y$ (update new value)

24

Cell semantics (2)



- {Exchange C X Y} binds x to the cell's content z and updates the new cell content to y
  - The exchange operation is atomic, which means the scheduler is guaranteed never to stop in the middle, it happens as one indivisible step
- We assume that environment E={C→c,X→x,Y→y,Z→z}

25

Ports

26

A port (named stream)

- To overcome the limitations of declarative programming, we add ports (named streams) to the language
- Ports have two operations:
  - P={NewPort S} % Create port P with stream S
  - {Send P X} % Add X to end of port P's stream
- This lets us do the client/server
  - With a million clients C_1 to $C_{1000000}$:
    Each client C_i does {Send P M_i} for each message it sends
  - The server reads the stream S, which contains all messages from all clients in some nondeterministic order

27

Port examples

- We create a port and do sends:

```
P={NewPort S}
{Browse S} % Displays _
{Send P a}  % Displays a|_
{Send P b}  % Displays a|b|_
```

- What happens if we do:

```
thread {Send P c} end
thread {Send P d} end
```

- What are the possible results of these two sends for all choices of the scheduler?

28

Port semantics (1)

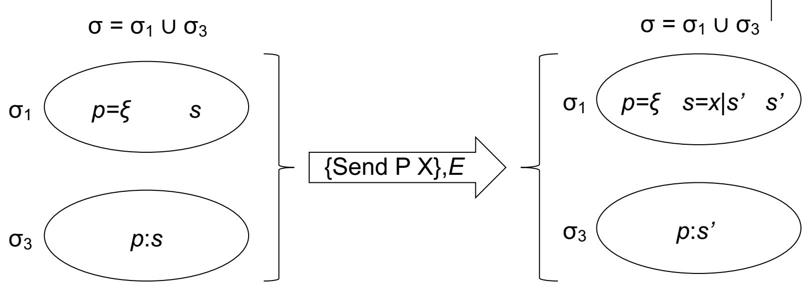
- Assume single-assignment store σ_1 with variables
- Assume a port store σ_3 that contains pairs of variables
  - (Remember σ_2 is the cell store we introduced before)

environment

- P={NewPort S}, {P→*p*, S→*s*}
  - Assume unbound variables p, $s \in \sigma_1$
  - Create fresh name ξ, bind p=ξ, add pair p:s to σ_3
- {Send P X}, {P→*p*, X→*x*}
  - Assume p=ξ, unbound variable $s \in \sigma_1$, p:$s \in \sigma_3$
  - Create fresh unbound variable s', bind s=x|s', update pair to p:s'

29

Port semantics (2)



- {Send P X} adds x to the end of the port's stream and updates the new end of stream
  - The send operation is atomic, which means the scheduler is guaranteed never to stop in the middle, so it happens as if it is one indivisible step
- We assume that environment E={P→p,X→x}

30

Cell + port semantics summary

- The full store $\sigma = \sigma_1 \cup \sigma_2 \cup \sigma_3$ has two different parts:
  - Single-assignment store (contains variables)
    σ_1 = {*t*, *u*, *v*, *x*=ξ, *y*=ζ, *z*=10, *w*=5}
  - Multiple-assignment store (contains pairs: cells and ports)
    $\sigma_2 \cup \sigma_3$ = {*x*:*t*, *y*:*w*}
- The multiple assignment store has two kinds of nondeclarative entities
  - Cells: mutable state
  - Ports: communication channel

31

Client/server with ports

- Assume port P={NewPort S}
- Client code: (any number of clients!)
  - {Send P M} sends message to server
- Server code:

```
proc {Server S}
    case S of M|T then
        (handle M)
        {Server T}
    end
end
```

32

Conclusions

33

Conclusions

- Declarative paradigms are the best but they cannot always be used
  - We investigate their limitations and how to overcome them
- Declarative paradigms are based on lambda calculus, which makes them confluent but they cannot interact with the real world
- To interact with the real world, we extend declarative paradigms with imperative concepts, like mutable state or communication channels
  - Mutable state (cells) leads to shared-state concurrency (Java)
  - Communication channels (ports) lead to message-passing concurrency (Erlang)
- Programs should use declarative paradigms as much as possible with as few imperative concepts as possible
  - The extensions should only be used in special cases, namely where interaction with the real world is needed

34

LINFO1131
Concurrent programming concepts

Lecture 6: Data abstractions

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

Overview

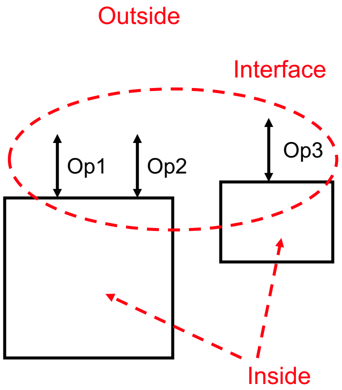
- Data abstractions and how to define them (lecture 6)
  - There are two fundamental kinds of data abstractions: objects and abstract data types
  - Furthermore, each of these kinds can either be declarative or nondeclarative, giving four kinds in all
  - We show how to define all these data abstractions using three concepts: static scope, higher-order programming, and unforgeable keys
- Abstract data types (ADTs)
  - An ADT consists of a set of values and a set of operations
- Objects
  - A single object represents both a value and a set of operations
- Four kinds of data abstraction
  - There are two axes: ADT or object, and stateless or stateful
  - This gives four ways to package a data abstraction!

Data abstractions

3

Definition of data abstraction

- A data abstraction is a part of a program that has an inside, an outside, and an interface in between
- The inside is hidden from the outside
  - All operations on the inside must pass through the interface, i.e., the data abstraction must use encapsulation
- The interface is a set of operations that can be used according to certain rules
  - Correct use of the rules guarantees that the results are correct
- The encapsulation must be supported by the programming language
  - We will see how the language can support encapsulation, that is, how it can enforce the separation between inside and outside



4

Building a data abstraction

- Assume your program uses a stack with the following implementation:

```
fun {NewStack} nil end
fun {Push S X} X|S end
fun {Pop S X} X=S.1 S.2 end
fun {IsEmpty S} S==nil end
```

- This implementation is not encapsulated!
  - It is implemented using lists that are not protected
    - A user can read stack values outside of the implementation
    - A user can create stack values outside of the implementation
- There is no way to guarantee that an unencapsulated stack will work correctly
  - The stack must be encapsulated → data abstraction

5

Two main kinds of data abstraction

- There are two fundamental kinds of data abstraction, namely objects and abstract data types
  - An object groups together value and operations in a single entity
  - An abstract data type keeps values and operations separate
- Some real world examples
  - A television set is an object: it can be used directly through its interface (on/off, channel selection, volume control)
  - Coin-operated vending machines are abstract data types: the coins and products are the values and the operations are the vending machines
- We will look at both objects and ADTs
  - Each has its own advantages and disadvantages

6

Abstract data types (ADTs)

7

Abstract data types

- An ADT consists of a set of values and a set of operations
- A common example: integers
  - Values: 1, 2, 3, …
  - Operations: +, -, *, div, …
- In most of the popular uses of ADTs, the values and operations have no state
  - The values are constants
  - The operations have no internal memory (they don't remember anything in between calls)

8

A stack ADT

- We can implement a stack as an ADT:
  - Values: all possible stacks and elements
  - Operations: NewStack, Push, Pop, IsEmpty
- The operations take (zero or more) stacks and elements as input and return (zero or more) stacks and elements as output
  - S={NewStack}
  - S2={Push S X}
  - S2={Pop S X}
  - {IsEmpty S}
- For example:
  - S={Push {Push {NewStack} a} b} returns the stack S=[b a]
  - S2={Pop S X} returns the stack S2=[a] and the top X=b

9

Unencapsulated implementation

- The stack we saw before is almost an ADT:
  - **fun** {NewStack} nil **end**
  - **fun** {Push S X} X|S **end**
  - **fun** {Pop S X} X=S.1 S.2 **end**
  - **fun** {IsEmpty S} S==nil **end**
- Here the stack is represented by a list
- But this is not a data abstraction, since the list is not protected
- How can we protect the list, and make this a true ADT?
  - How can we build an abstract data type with encapsulation?
  - We need a way to protect values

10

Encapsulation using a secure wrapper

- To protect the values, we will use a secure wrapper:
  - The two functions Wrap and Unwrap will "wrap" and "unwrap" a value
  - W={Wrap X}   % Given X, returns a protected version W
  - X={Unwrap W}   % Given W, returns the original value X
- The simplest way to understand this is to consider that Wrap and Unwrap do encryption and decryption using a shared key that is only known by them
- We need a new Wrap/Unwrap pair for each ADT that we want to protect, so we use a procedure that creates them:
  - {NewWrapper Wrap Unwrap} creates the functions Wrap and Unwrap
  - Each call to NewWrapper creates a pair with a new shared key

11

Secure encapsulation

- Building a secure encapsulation requires support from the programming language
  - Some languages do not support secure encapsulation (like C, C++, or Javascript)
- To support secure encapsulation, a necessary (but not sufficient) condition is that the language supports unbreakable abstraction boundaries
  - For example, in Java it is impossible to "look inside" a primitive type, to see its machine representation
- To support secure encapsulation for programmer-defined ADTs, the language must also support a form of unforgeable key

12

Building a secure wrapper (1)

- To support an unforgeable key, we add two concepts to the language
  - Unforgeable constants (called “names”)
    - “key” N={NewName} binds N to a new name value. Because the name is unforgeable, it cannot be printed or guessed!
  - Secure records (records with only a “.” operation, but no other operations – no Arity operation)
    - “lock” C={Chunk.new R} takes a record R and returns a secure record C (called “chunk” in Oz)

13

Building a secure wrapper (2)

- With names and chunks, we can define NewWrapper:

```
proc {NewWrapper Wrap Unwrap}
  Key={NewName} % Generate unique key
in
  fun {Wrap X}
    {Chunk.new w(Key:X)} % Lock X inside secure record
  end
  fun {Unwrap W} % Extract X from secure record
    W.Key
  end
end
```

14

Building a secure wrapper (3)

- Unwrap needs protection in case of wrong argument:

```
proc {NewWrapper Wrap Unwrap}
  Key={NewName}
in
  fun {Wrap X}
    {Chunk.new w(Key:X)}
  end
  fun {Unwrap W}
    try W.Key
    catch _ then raise error(unwrap(W)) end end
  end
end
```

Raise an exception if W is a wrong argument

15

Implementing the stack ADT

- Now we can implement a true stack ADT:

```
local Wrap Unwrap in
     {NewWrapper Wrap Unwrap}

     fun {NewStack} {Wrap nil} end
     fun {Push W X} {Wrap X|{Unwrap W}} end
     fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
     fun {IsEmpty W} {Unwrap W}==nil end
end
```

- How does this work? Look at the Push function: it first calls {Unwrap W}, which returns a stack value S, then it builds X|S, and finally it calls {Wrap X|S} to return a protected result

- Wrap and Unwrap are hidden from the rest of the program (static scoping)

16

Final remarks on ADTs

- ADT languages have a long history
  - The language CLU, developed by Barbara Liskov and her students in 1974, is the first
  - This is only a little bit later than the first object-oriented language Simula 67 in 1967
  - Both CLU and Simula 67 strongly influenced later object-oriented languages up to the present day
- ADT languages support a protection concept similar to Wrap/Unwrap
  - CLU has syntactic support that makes the creation of ADTs very easy
- Many object-oriented languages also support ADTs
  - For example, Java supports ADTs: Java integers are ADTs, and Java objects have some ADT properties

17

Objects

18

Objects

- A single object represents both a value and a set of operations
- Example interface of a stack object:

```
S={NewStack}
{S push(X)}
{S pop(X)}
{S isEmpty(B)}
```

- The stack value is stored inside the object S
- Example use of a stack object:

```
S={NewStack}
{S push(a)}
{S push(b)}
local X in {S pop(X)} {Browse X} end
```

19

Implementing the stack object

- Implementation of the stack object:

```
fun {NewStack}
      C={NewCell nil}
      proc {Push X} C:=X|@C end
      proc {Pop X} S=@C in C:=S.2 X=S.1 end
      proc {IsEmpty B} B=(@C==nil) end
in
      proc {$ M}
         case M of push(X) then {Push X}
         [] pop(X) then {Pop X}
         [] isEmpty(B) then {IsEmpty B} end
      end
end
```

Procedure dispatching

- Each call to NewStack creates a new stack object
- This represents the object as a one-argument procedure that does procedure dispatching: a case statement chooses the operation to execute
- Encapsulation is enforced by hiding the cell with static scoping

20

Stack as ADT and stack as object

- Here is the stack as ADT:

```
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push W X} {Wrap X|{Unwrap W}} end
   fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
   fun {IsEmpty W} {Unwrap W}==nil end
end
```

- Here is the stack as object:
  - This represents the object as a record that does record dispatching

```
fun {NewStack}
   C={NewCell nil}
   proc {Push X} C:=X|@C end
   proc {Pop X} S=@C in X=S.1 C:=S.2 end
   fun {IsEmpty} @C==nil end
in
   stack(push:Push pop:Pop isEmpty:IsEmpty)   } Record dispatching
end
```

- Any data abstraction can be implemented as an ADT or as an object

21

Final remarks on objects

- Objects are omnipresent in computing today
- The first major object-oriented language was Simula-67, introduced in 1967
  - It directly influenced Smalltalk (starting in 1971) and C++ (starting in 1979), and through them, most modern object-oriented languages (Java, C#, Python, Ruby, and so forth)
- Most modern OO languages are in fact data abstraction languages: they incorporate both objects and ADTs
  - And other data abstraction concepts as well, such as components and modules
  - The Java language has both ADTs (e.g., Integer) and objects

22

Four kinds of data abstraction

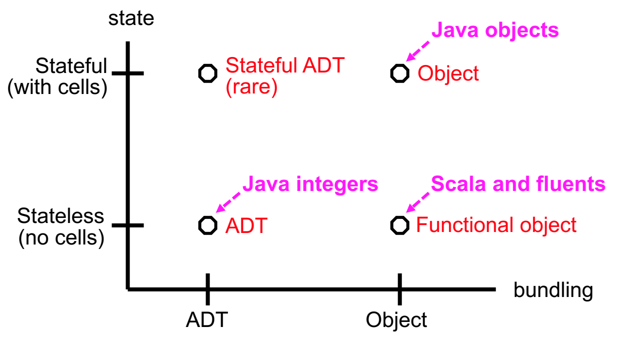
23

Four kinds of data abstraction

- We have seen two commonly used data abstractions:
  - Abstract data types (without mutable state: declarative)
  - Objects (with mutable state: nondeclarative)
- There are two other kinds of data abstractions
  - Abstract data types with state (stateful ADTs: nondeclarative)
  - Objects without state (functional objects: declarative)
- This gives four kinds in all
  - Let's take a look at the two additional kinds
  - And then we'll conclude this lesson on data abstraction

24

Four kinds of data abstraction



- Objects (with state) and ADTs (stateless) are in Java
- Functional objects are used in Scala and for big data
- Stateful ADTs are rarely used (so far!)

25

The two other data abstractions

- A functional object is possible
  - Functional objects are immutable; invoking an object returns another object with a new value
  - Functional objects are becoming more popular
- A stateful ADT is possible
  - Stateful ADTs were much used in the C language (although without enforced encapsulation, since it is impossible in C)
  - They are also used in other languages (e.g., classes with static attributes in Java)
- Let's take a closer look at how to build them

26

A functional object

- We can implement the stack as a functional object:

```
local
  fun {StackObject S}
     fun {Push E} {StackObject E|S} end
     fun {Pop S1}
        case S of X|T then S1={StackObject T} X end end
     fun {IsEmpty} S==nil end
  in stack(push:Push pop:Pop isEmpty:IsEmpty) end
in
  fun {NewStack} {StackObject nil} end
end
```

- This uses no cells and no secure wrappers. The simplest of all data abstractions, it only needs static scope and higher-order programming (which together guarantee unbreakable encapsulation).

27

Functional objects in Scala

- Scala is a hybrid functional-object language: it supports both the functional and object-oriented paradigms
- In Scala we can define an immutable object that returns another immutable object
  - For example, a RationalNumber class whose instances are rational numbers (and therefore immutable)
  - Adding two rational numbers returns another rational number
- Immutable objects are functional objects
  - The advantage is that they cannot be changed (the same advantage of any functional data structure)

28

A stateful ADT

- Finally, let us implement our trusty stack as a stateful ADT:

```
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   proc {Push S E} C={Unwrap S} in C:=E|@C end
   fun {Pop S} C={Unwrap S} in
      case @C of X|S1 then C:=S1 X end
   end
   fun {IsEmpty S} @{Unwrap S}==nil end
in
   Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)
end
```

- This uses both a cell and a secure wrapper. Note that Push, Pop, and IsEmpty do not need Wrap! They modify the stack state by updating the cell *inside* the secure wrapper.

29

Conclusions

30

Conclusions

- Data abstractions are the key to organizing programs
  - A data abstraction has an inside, an outside, and an interface between the two
  - The only way to access the inside is by using the interface
- Data abstractions come in four kinds, along two axes:
  - First axis: objects versus abstract data types (ADTs)
  - Second axis: declarative versus nondeclarative
- Building data abstractions
  - We show how to build the four kinds of data abstractions using static scoping and higher-order programming (which together guarantee unbreakable encapsulation)
  - For programmer-defined ADTs, the language must also support unforgeable keys
  - Mutable state can be used to build data abstractions that can model time and change

31

LINFO1131
Concurrent programming concepts

Lecture 7
Message passing
and multi-agent programming

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

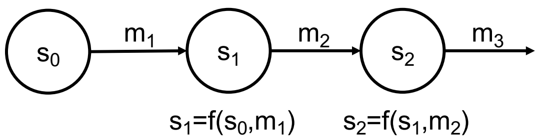
- Multi-agent programming with message passing is based on ports, which are a form of communication channel
  - Let us now look at multi-agent programming more closely!
- Port objects and active objects
  - A port object has an internal state, a thread, and a port. Sending a message causes the internal state to be updated according to a transition function.
  - An active object combines a port object and a class. This adds the advantages of object-oriented programming (polymorphism, inheritance) to port objects.
- Message protocols
  - Port objects and active objects can be used to define message protocols
- Flavius Josephus problem
  - This is a classic problem that we solve using both active objects and deterministic dataflow, to compare the two paradigms
- Multi-agent programming
  - We show how to write big programs using many port objects that talk to each other
- Lift control system
  - We give a worked-out example of a realistic multi-agent system

2

Port objects

3

Stateful port objects (Section 5.2)



- A stateful port object, also called stateful agent, has an internal memory s_i called its state
- The state is updated with each message received, which gives a state transition function:
  F: State $\times$ Msg $\mapsto$ State

4

Creating stateful port objects

- We define a generic function for stateful port objects:

```
fun {NewPortObject Init F}
    proc {Loop S State}
       case S of M|T then {Loop T {F State M}} end
    end
    P
in
    thread S in P={NewPort S} {Loop S Init} end
    P
end
```

5

Structure of Loop

- Does the Loop function ring a bell?

```
proc {Loop S State}
    case S of M|T then {Loop T {F State M}} end
end
```

- Loop starts from an initial state
- Loop successively applies F to the previous state and a message, to compute the next state
- The function F is a binary operation
- What is this?

6

Structure of Loop

- Does the Loop function ring a bell?

```
proc {Loop S State}
    case S of M|T then {Loop T {F State M}} end
end
```

- Loop starts from an initial state
- Loop successively applies F to the previous state and a message, to compute the next state
- The function F is a binary operation
- Of course! It is a Fold operation!

7

FoldL operation

- FoldL is an important higher-order operation:

```
fun {FoldL S F U}
    case S
    of nil then U
    [] H|T then {FoldL T F {F U H}}
    end
end
```

- $((\ldots(((u \text{ f } a_0) \text{ f } a_1) \text{ f } a_2) \ldots) \text{ f } a_{n-1})$

8

Fold is the heart of the agent

- We replace:
  ```
  thread S in P={NewPort S} {Loop S Init} end
  ```
- by:
  ```
  thread S in P={NewPort S} {FoldL S F Init} end
  ```
- Oops! There is a small bug…

9

Updated NewPortObject

- We define a generic function for stateful port objects:

```
fun {NewPortObject Init F}
    P Out
in
    thread S in P={NewPort S} Out={FoldL S F Init} end
    P
end
```

Important abstraction that combines FoldL, a port, and a thread

- Out is the final state when the agent terminates
  - It never terminates here, but in another definition it might (if the stream terminates with nil, then the port object terminates too)

10

Message protocols (refresher)

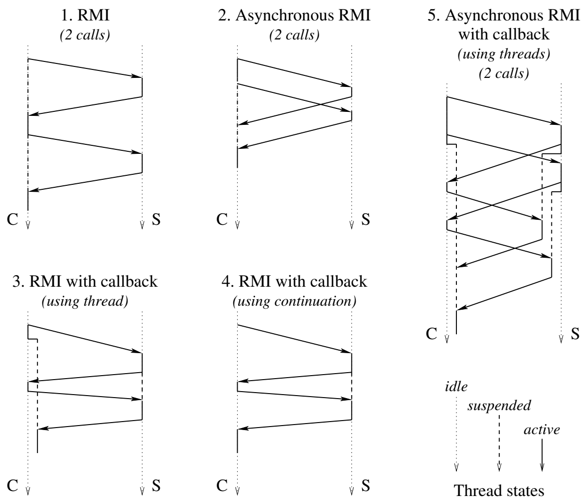
11

Message protocols (1)

- A message protocol is a sequence of messages between two or more parties that can be understood at a higher level of abstraction than individual messages
- Using port objects, we can implement some important message protocols
- We saw these protocols in the previous course LINFO1104
  - Explained in Section 5.3 of the course textbook

12

Message protocols (2)



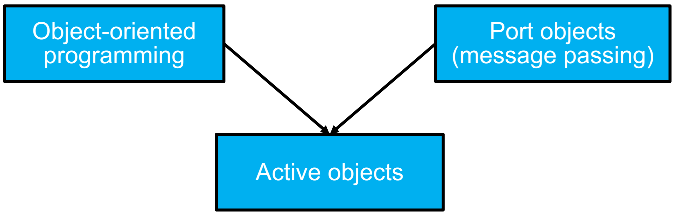
- We start with a simple RMI
- We then make it asynchronous and add callbacks
- The most complicated protocol shown here is asynchronous RMI with callback

13

Active objects

14

Active objects (Section 7.8)



- An active object is a port object whose behavior is defined by a class
- Active objects combine the abilities of object-oriented programming (including polymorphism and inheritance) and message-passing concurrency
- To explain active objects, we refresh your memory on object-oriented programming and we introduce classes in Oz

15

Classes and objects in Oz

- We saw objects in the course
- We now complete this explanation by introducing classes and their Oz syntax

```
class Counter
      attr i
      meth init(X)
          i := X
      end
      meth inc(X)
          i := @i + X
      end
      meth get(X)
          X=@i
      end
end
```

- Create an object:

  Ctr={New Counter init(0)}

- Call the object:

```
{Ctr inc(10)}
{Ctr inc(5)}
local X in
      {Ctr get(X)}
      {Browse X}
end
```

16

Defining active objects

- Active objects are defined by combining classes and port objects
  - Because objects have internal state (attributes), we can replace Fold by a for loop. Each object call corresponds to one execution of the state transition function.
- We return a one-argument procedure to make them look like standard Oz objects

```
fun {NewActive Class Init}
   Obj={New Class Init}
   P
in
   thread S in
      {NewPort S P}
      for M in S do {Obj M} end
   end
   proc {$ M} {Send P M} end
end
```

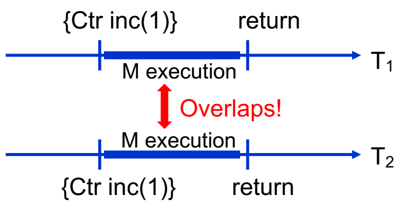
17

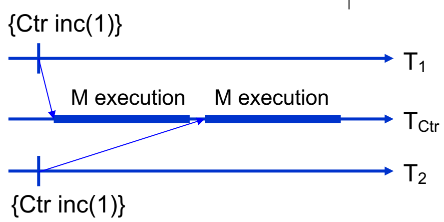
Passive objects and active objects

- We make a distinction between passive objects and active objects
- Standard objects in Oz (and in many other languages, such as Java and Python) are called passive objects
  - This is because they execute in the thread of their caller; they do not have their own thread
- This is in contrast to active objects, which have their own thread
- Let us compare passive and active objects!

18

Concurrency comparison





- Passive objects cannot be safely called from more than one thread
- The method executions can overlap, which leads to concurrency bugs

- Active objects are completely safe when called from more than one thread
- The method executions are executed sequentially in the active object's own thread

19

Passive objects are not concurrency-safe!

- The following code is buggy:

```
Ctr={New Counter init(0)}
thread {Ctr inc(1)} end
thread {Ctr inc(1)} end
local X in
    {Ctr get(X)}
    {Browse X}
end
```

- This can display 1! Why?
  - Look at the instruction i := @i +1
  - If the scheduler puts T1 to sleep after @i and before i:=, executes T2 fully, and then resumes T1

- The following code is correct:

```
Ctr={NewActive Counter init(0)}
thread {Ctr inc(1)} end
thread {Ctr inc(1)} end
local X in
    {Ctr get(X)}
    {Browse X}
end
```

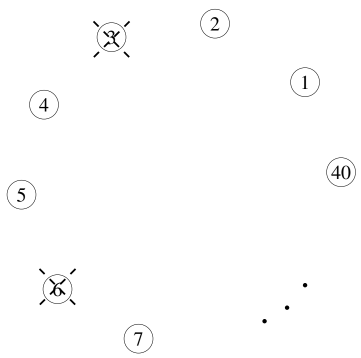
- This will always display 2
  - Because the two methods are executed sequentially by Ctr's thread

20

Flavius Josephus problem

21

Flavius Josephus problem (Section 7.8.3)



- Flavius Josephus was a Roman historian of Jewish origin. During the Jewish-Roman wars of the 1st century A.D., he was in a cave with fellow soldiers.
- Forty men decided to commit suicide by standing in a ring and counting off each third man. Josephus managed to place himself in the position of the last survivor.

22

Josephus protocol for N soldiers and K hops

- Message kill(X S) circulates around the ring, where X counts live objects traversed and S is the total number of live objects remaining
- Initially, kill(1 N) is given to the first object
- When an object receives kill(X S) it does the following:
  - If it is alive and S=1, then it is the last survivor (termination)
  - If it is alive and X mod K = 0, then it becomes dead and sends kill(X+1 S-1) to the next object
  - If it is alive and X mod K ≠ 0, then it sends kill(X+1 S) to the next object
  - If it is dead, then it forwards kill(X S) to the next object

23

Active objects versus deterministic dataflow

Let us compare active objects to deterministic dataflow. How do they compare in code complexity and efficiency?

Active object implementation	Deterministic dataflow implementation
• Each soldier is an active object (a passive object inside a thread)	• Each soldier is a stream object (a list function in a thread)
• Class Victim defines a kill(X S) method that implements the Josephus protocol	• Function Victim reads a stream of kill(X S) messages and outputs a new stream
• Initialization builds a ring with successor and predecessor pointers	• Initialization builds a ring of stream objects

24

Short-circuit protocol

Active object implementation

- We can optimize the active object version to remove dead victims from the ring (otherwise, a lot of time is lost in traversing them)
- We need to update the successor pointer of the predecessor node, and the predecessor pointer of the successor node
- It's a bit subtle because the modifications must be completed before the kill message traverses the ring
  - It depends on the FIFO property of the Send operation: do you understand why?

Deterministic dataflow implementation

- The deterministic dataflow version already does this optimization. It removes dead victims by replacing the recursive call {Victim Xr I} by Xr.
- Doing the recursive call to Victim means that the victim is still alive. Not doing the recursive call means the victim no longer exists.

25

Multi-agent program design

26

Multi-agent program design (Section 5.4)

- Programming with agents or concurrent components
  - The program is a collection of agents with internal state that send each other messages
  - With port objects, we can go beyond deterministic dataflow to design programs with nondeterministic behavior
    - Deterministic dataflow is a form of multi-agent programming that has deterministic behavior
    - Nondeterministic behavior is often needed when programs interact with the real world, e.g., they may have timing constraints that come from the real world (like in the client/server example), or else they interact with humans or machines
- We explain the basic principles of multi-agent programming
  - It is difficult because it must behave correctly for all possible interleavings of the agents (they execute concurrently!)
  - This means that it is very important to follow a rigorous design approach

27

Basic operations

- A concurrent component is a procedure with inputs and outputs
  - When invoked, the procedure creates a component instance, also known as an agent. In our examples, the agents will be made of port objects.
  - For example, the procedure FullAdder in the digital logic example: Calling {FullAdder X Y Ci S Co} creates a new full adder with three input streams X, Y, Ci and two output streams S and Co.
- Four basic operations
  - Instantiation: creating an instance of a component
  - Composition: building a new component out of other components
  - Linking: combining component instances by connecting inputs and outputs
    - Different kinds of links: **{one-source,many-source} × {one-shot,many-shot}**
  - Restriction: restricting visibility of inputs or outputs inside a component

28

Kinds of links

- We will use the following links in our examples
  - Other kinds of links are possible too and sometimes used in other systems
- One source (one sender)
  - One-shot (one message): Dataflow variable (single-assignment variable)
  - Many-shot (many messages): Stream (list that can be extended)
  - The link is deterministic: only one source can send a message
  - Deterministic dataflow only has one-source links
- Many source (many senders)
  - Many-shot: Port (named stream)
  - The link is nondeterministic: any source can send a message
  - Multi-agent programming uses many-source links (e.g., Erlang)
  - (Note that one-shot many-source links are not very useful)

29

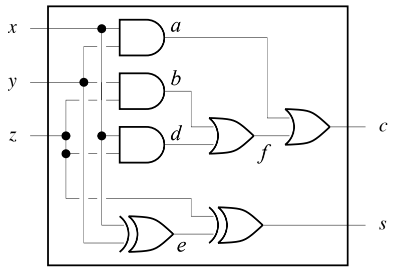
Example: Full Adder component

- The Full Adder from digital logic illustrates these operations:

```
proc {FullAdder X Y Z S C}
     A B D E F
in
     A={AndG X Y}
     B={AndG Y Z}
     D={AndG X Z}
     F={OrG B D}
     C={OrG A F}
     E={XorG X Y}
     S={XorG Z E}
end
```

Identifiers denote streams

Deterministic dataflow: no ports!

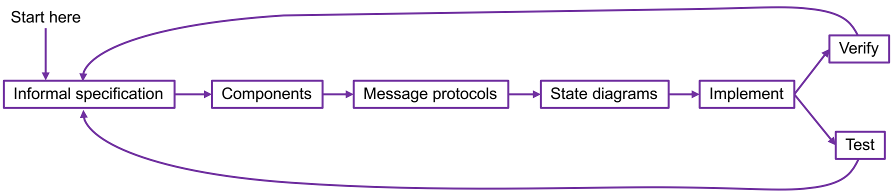


x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- A full adder adds three 1-bit binary numbers x, y, and z giving a sum bit s and carry bit c
- An n-bit adder can be built by connecting n full adders

30

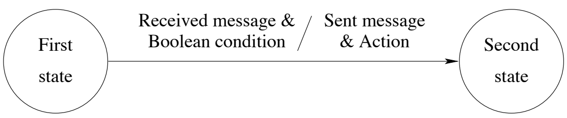
Design methodology



- Designing a multi-agent program is more difficult than designing a sequential program, because of all the possible interactions
  - That is why it is important to follow a rigorous methodology!

31

State transition diagrams



- A good way to design an agent is by enumerating the states it can be in and the messages it can send and receive
  - It is important to check that all messages can be handled in all states!
- A state transition diagram defines a finite state automaton. It has a finite set of states and a set of transitions between the states.
  - It evolves by doing transitions: when a message is received and a boolean condition is true, the state changes

32

Lift control system

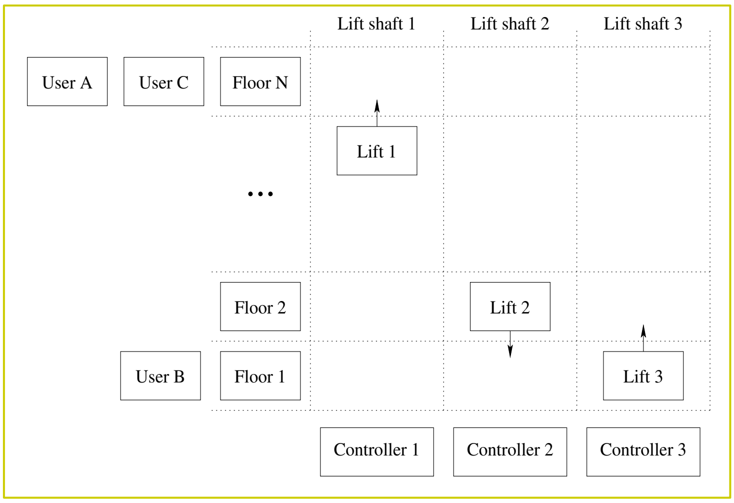
33

Lift control system (Section 5.5)

- We show the design and implementation of a lift control system
  - Similar to actual lift control systems in hotels and apartment buildings
  - We will follow our methodology
- We start with the informal specification
  - Overview of system and its embedding in the real world
  - We identify the components and message protocols
- We design the state diagrams
  - Three components: controller, floor, lift
  - Make sure that all possible messages can be handled
- We implement the program (write the code)
  - Simple translation of the state diagram
  - We test the program in various concurrent scenarios
  - We verify the program by reasoning about invariants

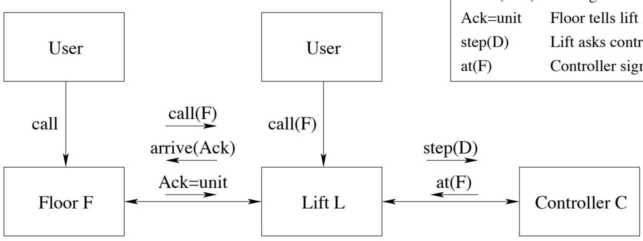
34

Lift control system

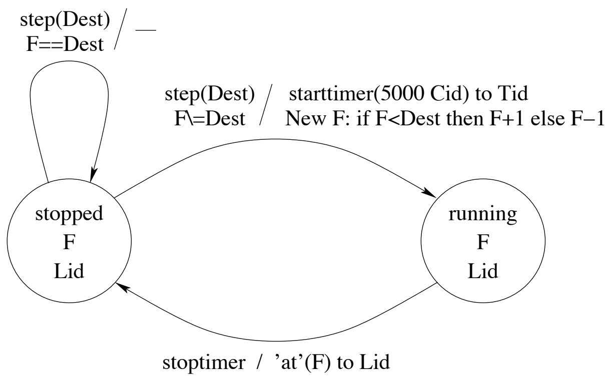


Lift components and their protocols

call	User presses button to call a lift
call(F)	Floor F calls a lift to itself
	User presses button to go to floor F
arrive(Ack)	Lift signals its arrival at the floor
Ack=unit	Floor tells lift it can leave now
step(D)	Lift asks controller to move one floor
at(F)	Controller signals successful move

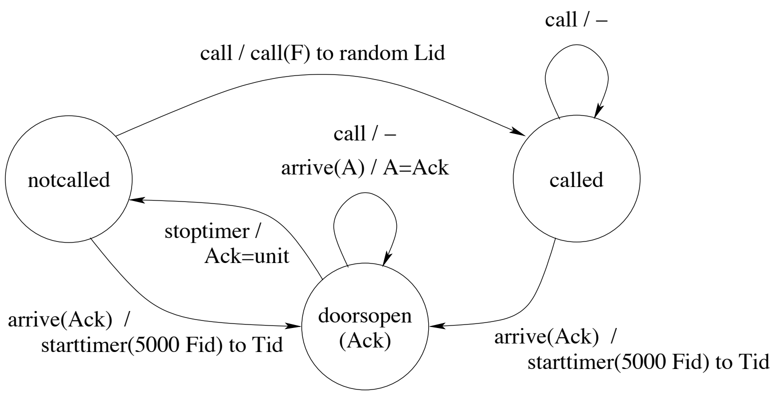


Controller state diagram



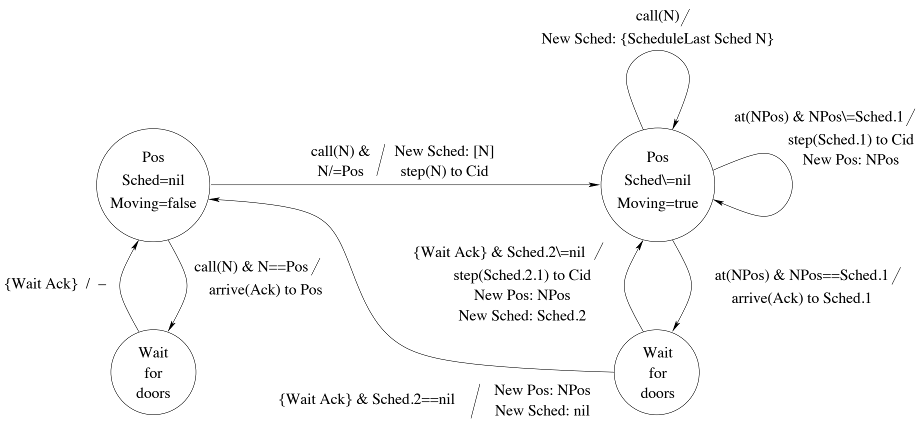
37

Floor state diagram



38

Lift state diagram



39

Implementation (see the code!)

- When all state diagrams are defined, we write the code by translating them into port objects

- In the lift control system we have three components:

  Fid={Floor Num Init Lifts}: creates floor Fid with number Num, initial state Init, and lifts Lifts (a tuple)

  Lid={Lift Num Init Cid Floors}: creates lift Lid with number Num, initial state Init, controller Cid, and floors Floors (a tuple)

  Cid={Controller Init}: creates a controller Cid with initial state Init

40

Improvements

- Procedure to create a building
  - Create all controller, lift, and floor agents
- Smarter scheduler
  - What happens if lift is at floor 1 and is called to floor 30, and during the movement there is another call to floor 15?
  - Rotating disk storage uses the smart scheduler

41

Testing and verification

- Both testing and verification are important!
  - Testing: run the system with many possible scenarios
  - Verification: reason about the system to prove invariants
- If a problem is found, then you go back to an earlier stage
  - Maybe all the way back to Informal Specification!
  - Follow the methodology again and fix the problem
  - Then do testing and verification again
- Example verifications (how would you prove them?)
  - Prove the following property: “The lift will always eventually stop at a floor that is part of the lift’s schedule”.
  - Prove the following property: “If a call(F) is sent to a lift, then the lift will eventually arrive at floor F”.

42

Three pillars of software development

Design | Test | Verify

- Design: our design methodology helps to ensure sound system structure
- Test: run the code on many different scenarios including corner cases
- Verify: prove properties of the system by reasoning on the state diagrams

- Leaving out any of the three is dangerous!

43

Conclusions

44

Conclusions

- We add ports (named streams) to overcome the limitations of deterministic dataflow
  - Ports allow nondeterministic many-to-one communication, which is not possible in deterministic dataflow
- With ports we can write multi-agent programs, which are programs made of concurrent agents that send messages to each other
  - An agent is implemented as a port object or an active object. Both have a port and a thread and an internal state that is updated when messages arrive. The port object's behavior is defined by a state transition function. The active object's behavior is defined by a class.
- We compare active objects and deterministic dataflow by programming the classic Flavius Josephus problem in both
  - You can see how the same protocol is implemented in both paradigms
- We explain how to build large multi-agent systems and we give an example of one such system, namely a lift control system

45

```
% LINFO1131
% Advanced Programming Language Concepts

% Lecture 8 (Nov. 22, 2023)

% Message-passing concurrency and multi-agent programming

% - Port objects and active objects
% - Flavius Josephus problem: comparing active objects
%   and deterministic dataflow
% - Lift control system: example of a realistic
%   multi-agent system

%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%

% 1. Port objects and active objects

% 1.1. Port object with internal state
declare
fun {NewPortObject Init Fun}
   P
in
   thread Sin Sout in
      {NewPort Sin P}
      {FoldL Sin Fun Init Sout}
   end
   P
end

% 1.2. Port object without internal state
declare
fun {NewPortObject2 Proc}
   P
in
   thread Sin in
      {NewPort Sin P}
      for Msg in Sin do {Proc Msg} end
   end
   P
end

% 1.3. Active object (port object with a class)
declare
fun {NewActive Class Init}
   Obj={New Class Init}
   P
in
   thread S in
      {NewPort S P}
      for M in S do {Obj M} end
   end
   proc {$ M} {Send P M} end
end


%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%

% 2. Flavius Josephus problem

% We define two versions of this problem:
% - Active object version with a class definition
% - Deterministic dataflow version with streams
```

```
% We can make a dataflow version because the
% Flavius Josephus problem is deterministic.
% Compare the two!  Which is longest, which is shortest!

% 2.1 Active object version of Flavius Josephus
declare
class Victim
   attr ident alive step last succ
   meth init(I K L)
      alive:=true step:=K last:=L ident:=I
   end
   meth setSucc(S) succ:=S end
   meth kill(X S)
      if @alive then
          if S==1 then
             @last=@ident
          elseif X mod @step ==0 then
             alive:=false
             {@succ kill(X+1 S-1)}
          else
             {@succ kill(X+1 S)}
          end
      else
          {@succ kill(X S)}
      end
   end
end

declare
fun {Josephus N K}
   A={NewArray 1 N null}
   Last
in
   % N objects
   for I in 1..N do
      A.I:={NewActive Victim init(I K Last)}
   end
   % Connect them into a ring
   for I in 1..(N-1) do
      {A.I setSucc(A.(I+1))}
   end
   {A.N setSucc(A.1)}
   {A.1 kill(1 N)}
   Last
end

{Browse {Josephus 5 2}}

{Browse {Josephus 40 3}}

{Browse {Josephus 1000 100}}

%%%%%%%%%%%%%%%%%%%

% 2.2 Optimized active object version that removes dead victims from ring
% Also known as "short-circuit" version

declare
class Victim2
   attr ident alive step last succ pred
   meth init(I K L)
      alive:=true step:=K last:=L ident:=I
   end
   meth setSucc(S) succ:=S end
```

```
   meth setPred(P) pred:=P end
   meth kill(X S)
      if @alive then
         if S==1 then
            @last=@ident
         elseif X mod @step ==0 then
            alive:=false
            {@pred setSucc(@succ)} % The order of messages is critical
            {@succ setPred(@pred)} % Kill must encounter a correct ring
            {@succ kill(X+1 S-1)}  % This works because of FIFO property
         else
            {@succ kill(X+1 S)}
         end
      else
         {@succ kill(X S)}
      end
   end
end

declare
fun {Josephus2 N K}
   A={NewArray 1 N null}
   Last
in
   % N objects
   for I in 1..N do
      A.I:={NewActive Victim2 init(I K Last)}
   end
   % Connect them into a ring
   for I in 1..(N-1) do
      {A.I setSucc(A.(I+1))}
   end
   {A.N setSucc(A.1)}
   % Correctly set the predecessors
   for I in 2..N do
      {A.I setPred(A.(I-1))}
   end
   {A.1 setPred(A.N)}
   {A.1 kill(1 N)}
   Last
end

{Browse {Josephus2 5 2}}
{Browse {Josephus2 1000 100}}

%%%%%%%%%%%%%%%%%%%

% 2.3 Deterministic dataflow version of Flavius Josephus
% Code is very compact: streams compacter than explicit message passing
% Only possible because Flavius Josephus is a deterministic algorithm
% This version does the short-circuit optimization

% Exercise: try to make each line of versions 2.2 and 2.3 correspond.

declare
fun {Pipe Xs L H F}
   if L>H then Xs else {Pipe {F Xs L} L+1 H F} end
end

declare
fun {Josephus3 N K}
   fun {Victim Xs I}
      case Xs of kill(X S)|Xr then
         if S==1 then Last=I nil
```

```
         elseif X mod K == 0 then
            kill(X+1 S-1)|Xr
         else
            kill(X+1 S)|{Victim Xr I}
         end
      [] nil then nil end
   end
   Last Zs
in
   Zs={Pipe kill(1 N)|Zs 1 N
       fun {$ Is I} thread {Victim Is I} end end}
   Last
end

{Browse {Josephus3 5 2}}
{Browse {Josephus3 40 3}}
{Browse {Josephus3 1000 100}}

%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%

% 3. Lift control system

% This is an example of a realistic multi-agent system.

% Each kind of port object is first defined by drawing a complete state
% diagram. Then the state diagram is translated into code.  The hard
% part is defining the state diagram.  Translating into code is easy!
% The code has two nested case statements, one case statement for the
% current state and a second case for the message that arrives.  The
% result is the new state.

% 1.1. Port object with internal state
declare
fun {NewPortObject Init Fun}
   P
in
   thread Sin Sout in
      {NewPort Sin P}
      {FoldL Sin Fun Init Sout}
   end
   P
end

% 1.2. Port object without internal state
declare
fun {NewPortObject2 Proc}
   P
in
   thread Sin in
      {NewPort Sin P}
      for Msg in Sin do {Proc Msg} end
   end
   P
end

% Send starttimer(T Pid) message, return message sent after T milliseconds
declare
fun {Timer}
   {NewPortObject2
    proc {$ Msg}
       case Msg of starttimer(T Pid) then
           thread {Delay T} {Send Pid stoptimer} end
       end
```

```
    end}
end

% 3.1 Controller agent
declare
fun {Controller Init}
   Tid = {Timer}
   Cid = {NewPortObject Init
           fun {$ state(Motor F Lid) Msg}
              case Motor
              of running then
                  case Msg
                  of stoptimer then
                     {Send Lid 'at'(F) }
                     state(stopped F Lid)
                  end
              [] stopped then
                  case Msg
                  of step(Dest) then
                     if F==Dest then
                        state(stopped F Lid)
                     elseif F < Dest then
                        {Send Tid starttimer(1000 Cid)}
                        state(running F+1 Lid)
                     else
                        {Send Tid starttimer(1000 Cid)}
                        state(running F–1 Lid)
                     end
                  end
              end
           end}
in Cid end

% 3.2 Floor agent
declare
fun {Floor Num Init Lifts}
   Tid= {Timer}
   Fid= {NewPortObject Init
          fun {$ state(Called) Msg}
             case Called
             of notcalled then Lran in
                case Msg
                of arrive(Ack) then
                    {Browse 'Lift at floor '#Num#': open doors'}
                    {Send Tid starttimer(2000 Fid)}
                    state(doorsopen(Ack))
                [] call then
                    {Browse 'Floor '#Num#' calls a lift!'}
                    Lran=Lifts.(1+{OS.rand} mod {Width Lifts})
                    {Send Lran call(Num)}
                    state(called)
                end
             [] called then
                case Msg
                    of arrive(Ack) then
                       {Browse 'Lift at floor'#Num#': open doors'}
                       {Send Tid starttimer(2000 Fid)}
                       state(doorsopen(Ack))
                    [] call then
                       state(called)
                    end
             [] doorsopen(Ack) then
                    case Msg
                    of stoptimer then
```

```
                      {Browse 'Lift at floor '#Num#': close doors'}
                      Ack=unit
                      state(notcalled)
                   [] arrive(A) then
                      A = Ack
                      state(doorsopen(Ack))
                   [] call then
                      state(doorsopen(Ack))
                   end
               end
            end}
in Fid end

% 3.3 Lift agent (with schedule function)
declare
fun {ScheduleLast L N}
   if L\=nil andthen {List.last L} == N then L
   else {Append L [N]} end
end

fun {Lift Num Init Cid Floors}
   {NewPortObject Init
    fun {$ state(Pos Sched Moving) Msg}
       case Msg
       of call(N) then
           {Browse 'Lift '#Num#' needed at floor '#N}
           if N==Pos andthen {Not Moving} then
              {Browse 'At '#N#' floor!'}
              {Wait {Send Floors.Pos arrive($)}}
              state(Pos Sched false)
           else Sched2 in
              Sched2={ScheduleLast Sched N}
              if {Not Moving} then
                  {Send Cid step(N)} end
              state(Pos Sched2 true)
           end
       [] 'at'(NewPos) then
           {Browse 'Lift '#Num#' at floor '#NewPos}
           case Sched
           of S|Sched2 then
              if NewPos==S then
                  {Wait {Send Floors.S arrive($)}}
                  if Sched2==nil then
                     state(NewPos nil false)
                  else
                     {Send Cid step(Sched2.1)}
                     state(NewPos Sched2 true)
                  end
              else
                  {Send Cid step(S)}
                  state(NewPos Sched Moving)
              end
           end
       end
    end}
  end

% 3.4 Building with FN floors and LN lifts
declare
proc {Building FN LN ?Floors ?Lifts}
   Lifts={MakeTuple lifts LN}
   for I in 1..LN do Cid in
      Cid= {Controller state(stopped 1 Lifts.I)}
      Lifts.I={Lift I state(1 nil false) Cid Floors}
```

```
   end
   Floors={MakeTuple floors FN}
   for I in 1..FN do
      Floors.I={Floor I state(notcalled) Lifts}
   end
end

/*

% Exercise: run the lift control system with various messages
declare F L in
{Building 10 2 F L}

{Send F.9 call}

{Delay 300}
{Send F.5 call}
{Send L.1 call(4)}
{Send L.2 call(1)}
%{Delay 5000}
%{Send L.1 call(3)}
%{Send L.2 call(3)}

*/

%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%
```

LINFO1131
Concurrent programming concepts

Lecture 8
Robust multi-agent programming in Erlang

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Introduction
  - Erlang performance graphs for concurrency and message passing
- Part I: primitive concepts
  - Pure functional core
  - Spawning and message passing
  - Process linking
  - Dynamic code update
- Part II: building robust systems
  - Erlang philosophy
  - Behaviors, stable storage, and testing
  - Building a generic server
    - Avoiding race conditions, synchronous and asynchronous calls
  - Building a generic supervisor
- Conclusions

2

Erlang overview

- Erlang was developed by Ericsson for telecommunications in 1986 (Java is from 1991)
  - It is released as OTP (Open Telecom Platform) with a full set of libraries
  - It is supported by Ericsson, the Erlang Ecosystem Foundation, and a user community (www.erlang.org)
- Erlang programs consist of lightweight “processes”, which are port objects that communicate using asynchronous FIFO message passing
  - Erlang processes share nothing: all data is copied between them
  - Erlang processes receive messages through a mailbox that is accessed by pattern matching. Messages can be received out of order if they match.
- Erlang supports building reliable long-lived distributed systems
  - Erlang language and Erlang/OTP platform provide all the primitives and techniques to support the “let it crash” philosophy, e.g., using failure linking and supervisor trees
  - Ericsson AXD 301 ATM switch with 1.7 million lines of Erlang claims 99.9999999% availability (one may doubt the number of 9’s, but the system is extremely available!)

3

Bibliography

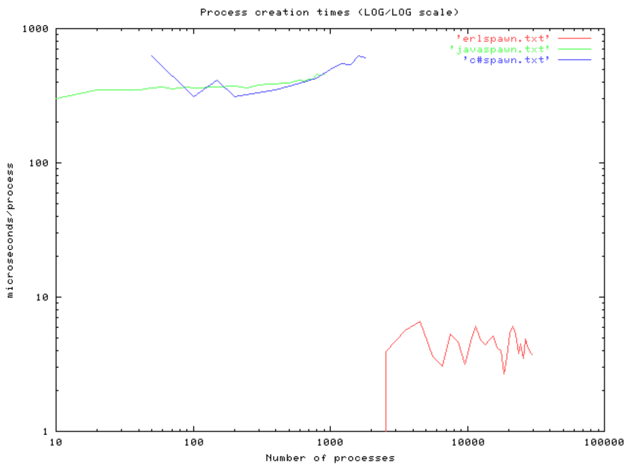
- Joe Armstrong, Concurrency Oriented Programming in Erlang, Talk slides, Nov. 2002.
- Joe Armstrong, Making Reliable Distributed Systems in the Presence of Software Errors, Ph.D. dissertation, KTH, Dec. 2003.
- Joe Armstrong, Programming Erlang: Software for a Concurrent World, The Pragmatic Bookshelf, 2007.
- Staffan Blau and Jan Rooth. AXD 301–A New Generation ATM Switching System, Ericsson Review No. 1, 1998.
- Francesco Cesarini and Steve Vinoski. Designing for Scalability with Erlang/OTP, O’Reilly, 2016.
- Ericsson AB. Erlang (Condensed), Talk slides.
- Ericsson AB. Erlang/OTP System Documentation Version 10.7, March 2020.
- Fred Hébert. Learn You Some Erlang for Great Good, learnyousomeerlang.com, 2020.
- Ulf Wiger. Four-fold Increase in Productivity and Quality, March 2001.

4

Erlang performance for concurrency and message passing

5

Erlang process creation times

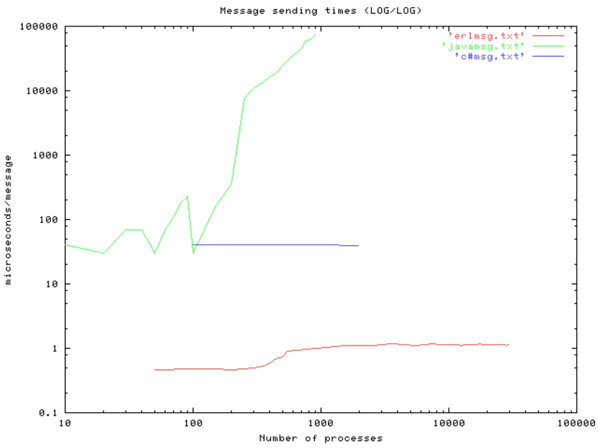


From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- We compare process creation times for Erlang, Java, and C#
- These numbers were measured in 2002; relative times should be similar today

6

Erlang message sending times

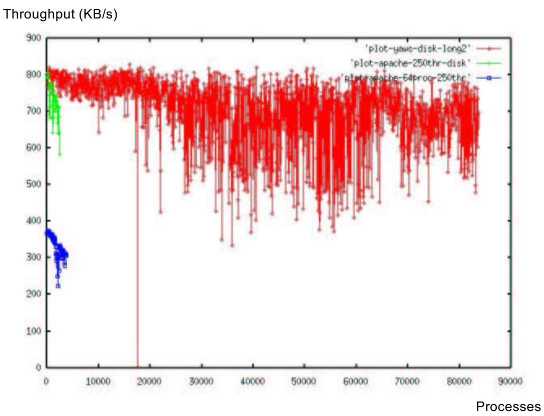


From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- We compare message sending times for Erlang, Java, and C#
- These numbers were measured in 2002; relative times should be similar today

7

Use case: web server



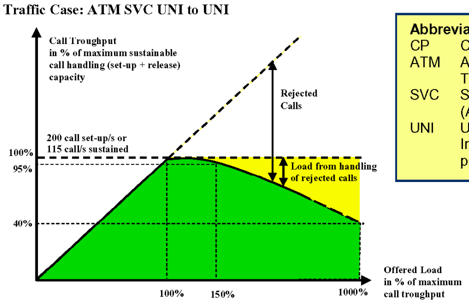
From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- Throughput versus number of processes for Web servers
  - Red = yaws (Yet Another Web Server, in Erlang on NFS)
  - Green = apache (local disk)
  - Blue = apache (NFS)
- Yaws: 800 KB/s up to 80,000 processes
- Apache: crashes at around 4,000 processes

8

Use case: AXD301 Erlang-based ATM switch

Call Handling Throughput for one CP - AXD 301 release 3.2
Traffic Case: ATM SVC UNI to UNI



From Ulf Wiger, Four-fold Increase in Productivity and Quality, 2001.

Abbreviations:

CP	Control Processor
ATM	Asynchronous Transfer Mode
SVC	Switched Virtual (ATM) Channel
UNI	User-Network Interface signaling protocol

- The AXD 301 is a general-purpose high-performance ATM switch from Ericsson
  - The AXD 301 is built using Erlang OTP supplemented with C and Java
- Throughput drops linearly when overloaded
  - 95% throughput at 150% load, descending to 40% throughput at 1000% sustained load
- AXD 301 release 3.2 has 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java
  - In addition, Erlang/OTP at that time had 240KLOC Erlang, 460KLOC C/C++, 15KLOC Java

9

Part I: Primitive concepts

10

Pure functional core

- Within a process, Erlang runs as a pure functional language
  - All variables are single assignment (bound when they are declared)
  - Functions are values with lexically scoped higher-order programming
  - Pattern matching used in **case** and **if** statements (and **receive**)
- All data structures are symbolic values
  - Integers (arbitrary precision), floats, atoms (symbolic constants)
  - Lists [george,paul,john,ringo] and tuples {Key,Val,L,R}
  - Strings are lists of ASCII codes (integers)
  - Binary vectors (used for protocol computations)

11

Dynamic typing versus static typing

- Erlang is a strongly typed language: types enforced by the language
  - Many languages are strongly typed, such as Java, Scheme, Haskell, and Prolog
  - Weakly typed languages, e.g., C and C++, allow access to a type's internal representation
- Strongly typed languages can be dynamically or statically typed
  - Erlang is a dynamically typed language because variables can be bound to entities of any type
  - In a statically typed language, variable types are known at compile time
- Static typing allows catching some program errors at compile time
  - However, this does not mean that statically typed programs are more resilient
    - Static typing allows catching many "surface errors" but does not help with "deep errors"
  - Well-written Erlang programs are among the most resilient software artifacts ever built, because Erlang provides adequate mechanisms for deep errors

12

An Erlang module

- The source code of an Erlang program is organized in modules:

```
-module(math).
-export([areas/1]).
-import(lists, [map/2]).

areas(L) -> lists:sum(map(fun(I) -> area(I) end, L)).

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

- Modules import and export, giving a dependency graph of modules

13

Message passing

14

Creating processes and sending messages

- Any process can create another by calling spawn
  - Pid = spawn(Fun) : function Fun defines behavior, Pid is process name
  - Fun may be anonymous or named
    - `fun(args) -> expr end`
    - `fun name/arity`
- Process name Pid is a unique constant that identifies the process
- Messages can be sent to the process using the process name
  - Pid ! Message
  - Messages are sent asynchronously and all data in messages is copied
  - Messages can be received by the **receive** statement

Receiving messages

- Each process has a mailbox that contains an ordered list of messages received by the process
- Messages are extracted from the mailbox using the **receive** statement
  - The **receive** uses pattern matching to remove the first message that matches
  - **receive**

```
receive
   pattern1 when guard1 -> expr1;
   pattern2 when guard2 -> expr2;
   …
   patternN when guardN -> exprN
end
```

Send and receive

```
Pid ! Message,
…

receive
      Message1 ->
        Actions1;
      Message2 ->
        Actions2;
      …
      after Time ->
        TimeOutActions
end
```

17

Receive mailbox semantics

- When a process executes **receive**:
  - If the mailbox is empty, **receive** blocks and waits for a message
  - If the mailbox is not empty, it takes the first message and tries patterns in order starting from the first, if it finds a matching pattern it executes the corresponding code
  - If no pattern matches, the **receive** blocks and waits for the next message
    - Unmatched messages remain in the mailbox to be removed by future **receive**
    - This allows different parts of a process to treat different kinds of messages
    - Messages can be removed out-of-order (in a different order from their arrival)
    - Take care that messages do not stay in the mailbox forever (memory leak)

- Patterns are symbolic data structures containing variable identifiers and guards are simple built-in tests

18

Process registering

- A process Pid can be registered with a name, which is an atom, to make the process globally available
  - This is important to keep interfaces unchanged, even as processes crash and are restarted
- register(Atom, Pid) : give Pid the global name Atom
- unregister(Atom) : remove the registration for Atom
- whereis(Atom) -> Pid | undefined : returns the Pid of a registered process, or undefined if no such process exists
- registered() -> [Atom :: atom()] : returns a list of all registered processes

19

Process linking

20

Process linking

- Process linking is a primitive operation important for fault tolerance
- Two processes can be linked together
  - Process Pid1 calls link(Pid2) or conversely; linking is bidirectional
- Process termination: send exit signal
  - A process terminates with an exit reason, sent as a signal to all linked processes
  - When a process terminates normally, the exit reason is the atom normal, otherwise when there is a run-time error, the exit reason is {Reason,Stack}
- Propagating process termination: transitive by default (“link set”)
  - The default behavior when a process receives an exit signal with reason other than normal is to terminate and to send exit signals with the same reason to its linked processes
  - A process can be set to trap exit signals by calling process_flag(trap_exit,true)
    - A received exit signal is then transformed into the message {‘EXIT’, FromPid, Reason}, which is put into the mailbox of the process

21

Using process linking

- If a process throws an exception that is not caught at the top level, then the process terminates and broadcasts its exit signal to all linked processes
- A few additional operations are defined to manage this:
  - exit(Pid,Why): send an exit signal to Pid without terminating
  - exit(Pid,kill): send an unstoppable exit to process Pid

```
start() -> spawn(fun go/0).

go() ->
     process_flag(trap_exit, true),
     loop().

loop() ->
     receive
         {'EXIT',Pid,Why} -> …
     end.
```

22

Process monitoring

- Process monitor is an asymmetric version of linking
  - No resemblance to monitor concept in shared-state concurrency!
- For example, in a client/server, if the server crashes we want to kill the clients, but if a client crashes we do not want to kill the server
- If process Pid1 executes:

  Ref = erlang:monitor(process, Pid2)

- Then if Pid2 dies with exit reason Why, then Pid1 will be sent a message {'DOWN',Ref,process,B,Why}

23

Distributed Erlang

- A distributed Erlang system consists of a number of Erlang runtime systems, called *nodes*, communicating with each other
- Message passing between processes at different nodes, as well as links and monitors, is transparent when using Pids
- Nodes can spawn processes on other nodes using spawn(Node,M,F,A)
- Registered names are local to each node: both node and name must be specified when sending messages using registered names
- The first time the name of a node is used, a connection attempt is made to the node; connections are made transitively giving a fully connected mesh by default (recent versions of Erlang allow more scalable connection topologies)

24

Dynamic code update

25

Dynamic code update

- In a real-time system, we would like to change the code without stopping the system
  - Some systems are never supposed to be stopped, e.g., the X2000 satellite control system developed by NASA
  - Hot code changing is difficult in a monolithic programming system, however, Erlang makes it possible because processes are independent (no sharing)
- Erlang allows each module to have two versions of code
  - All new processes will be dynamically linked to the latest version
  - If the code is changed, then processes can choose to continue with the old code or to use the new code
  - The choice is determined by how the code is called

26

Dynamic code update example

- Call the new version (if available)

```
-module(m).

loop(Data, F) ->
      receive
       {From,Q} ->
            {Reply,Data1}=F(Q,Data),
            m:loop(Data1, F)
      end.
```

Use new version

- This call is only used by libraries that manage upgrading of application releases

- Keep calling the old version:

```
-module(m).

loop(Data, F) ->
      receive
       {From,Q} ->
            {Reply,Data1}=F(Q,Data),
            loop(Data1, F)
      end.
```

Use old version

- This call is used for all the normal execution in one version

27

Fine-grained code update

- There is a second way to do code update using only higher-order functions
- This does not require any system support since it is based on passing functions as arguments
- This approach can be used for individual processes inside a module; note that the module code is not updated!

Loop code

```
server(Fun, Data) ->
      receive
           {new_fun,Fun1} ->
                server(Fun1,Data);
           {rpc,From,ReplyAs,Q} ->
                {Reply,Data1} =
                       Fun(Q,Data),
                From!{ReplyAs,Reply},
                server(Fun, Data1)
      end.
```

```
rpc(A,B) ->
      Tag=new_ref(),
      A!{rpc,self(),Tag,B},
      receive
             {Tag,Val} -> Val
      end.
```

28

Part II:
Building robust systems

29

Erlang philosophy

- How can we make robust software?
  - Popular languages (e.g., Java and Python) are inadequate
- Principles of robust software (from Joe Armstrong's Ph.D. thesis)
  - Errors cannot be fully eliminated, therefore they must be handled (both hardware and software errors)
  - Software components are the units of failure: errors occurring in one will not affect others ("strong isolation")
  - Software should be fail-fast: function correctly or stop quickly
  - Failure should be detectable by remote components
  - Software components share no state, but send messages

30

Erlang "slogans"

- "Let it crash", "If you can't do your job, crash"
  - Instead of trying to fix things when errors happen, which leads to a large number of complicated states, instead map everything to one simple state, namely "crashed"
- "Let some other process do error recovery"
  - Both hardware and software errors can occur and trying to solve them in the process makes things complicated. It is better to detect and handle any error, either in hardware or software, elsewhere.
- "Do not program defensively"
  - Defensive programming means to add checks in the program. This is not productive since it makes the program complicated (in particular, what do you do when a check fails?) and it will not remove all errors. Errors will still occur and still need to be handled. The best way is to map all errors no matter how bizarre to a single fault state, namely "crashed".

31

Erlang/OTP systems

- Erlang/OTP supports a hierarchy of systems:
  - Release: Contains all the information necessary to build and run a system, including a software archive and a set of procedures for installation (including upgrading without stopping)
  - Application: Contains all the software necessary to run a single application, not the entire system. Releases are often composed of multiple applications that are largely independent of one another, or that are hierarchically dependent.
  - Behavior: A set of processes that together implement a concurrency pattern
    - A notable behavior is supervisor: a tree of processes whose purpose is to monitor behaviors and each other and restart them when necessary
  - Worker: A process that is an instance of a behavior, usually instances of gen_server, gen_event or gen_fsm

32

Using behaviors to abstract concurrency and fault tolerance

- Program code can be divided into “hard” and “easy” modules
  - The hard modules should be few and written by expert programmers
  - The easy modules should be many and written by regular programmers
- Concurrency and fault tolerance are hard to implement
  - Behaviors are generic components that hide concurrency and fault tolerance
  - Behaviors are “hard modules” that are part of the Erlang/OTP platform
- The Erlang/OTP platform provides library support for many important behaviors
  - See Erlang/OTP System Documentation, Ericsson, Version 10.7, March 15, 2020

33

Standard Erlang/OTP behaviors

- The Erlang/OTP platform provides the following five standard behaviors:
  - Generic server (gen_server): to build client/server architectures with registration, start/stop, timeouts, state management, synchronous/asynchronous calls, error handling
  - Generic event handler/manager (get_event): event handlers, such as loggers, to respond to a stream of events, handling them and sending notifications
  - Generic finite state machine (gen_fsm): applications (e.g., protocol stacks) can be modeled as finite state machines, which provides a set of rules State × Event → Actions × State
  - Application: a component that can be started and stopped as a unit, and can be reused in other systems
  - Supervisor: the generic toolkit for implementing fault tolerance with supervisor hierarchies
- These behaviors hide most of the complexity of each concept, in particular both concurrency and fault tolerance are vastly simplified using behaviors
  - All non-supervisor behaviors are designed to be pluggable into a supervisor hierarchy

34

Erlang stable storage

- Resilient systems need stable storage to survive crashes
  - When a process is restarted by a supervisor, it uses the stable storage to start in a consistent state
  - Stable storage and supervisor trees are the two pillars of Erlang's resilience
- Erlang provides three levels of stable storage
  - ETS (Erlang Term Storage): Efficient in-memory storage for arbitrary Erlang terms, with limited size and concurrency abilities
  - DETS (Disk-based ETS): Same abilities as ETS, but stored on disk
    - ETS and DETS are limited to single nodes (single Erlang virtual machine)
  - Mnesia: A transactional database that is built on top of ETS and DETS and allows making a balance between ETS efficiency and DETS persistence
    - Mnesia supports distribution and replication on multiple nodes

35

Testing

- It's not enough to design for resilience, testing is essential
- Erlang provides a spectrum of practical and powerful testing tools
  - EUnit: Unit testing framework, including test generators (using higher-order programming to generate new tests) and fixtures (scaffolding around tests)
    - Supports test-driven development (TDD)
    - Good for testing modules and libraries
  - Common Test: Full-featured system testing framework
    - Test groups: allows running tests in parallel or in random order, for race conditions
    - Test suites: for handling dependencies between applications when testing
    - Test specifications including simulating abnormal termination (fault injection)
  - Dialyzer: Dynamic type checker based on success types
    - Will not make a proof of correctness, but any type error it finds is a real error

36

Building a generic server

37

Constructing a behavior

- An Erlang behavior is like a design pattern for a concurrent multi-agent system
- OTP encapsulates the most commonly used patterns in a set of generic library modules called OTP behaviors
- Behaviors abstract away all the tricky aspects and borderline conditions, through a solid well-tested reusable code base
- Let's see how this works for the gen_server!

38

Erlang process loop

- A typical Erlang server loop looks like this:

```
start(Args) -> spawn(server, init, [Args]).

init(Args) ->
   State=initialize_state(Args),
   loop(State).

loop(State) ->
   receive
      {handle, Msg} ->
         NewState=handle(Msg,State),
         loop(NewState);
      stop -> terminate(State)
   end.

terminate(State) -> clean_up(State).
```

We will split this loop into a generic part and a specific part

39

Towards a generic client/server

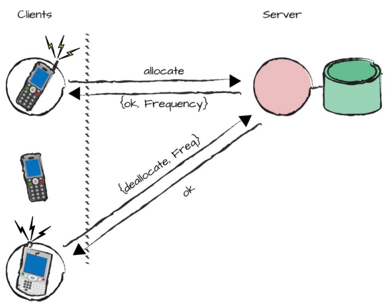
- Generic part
  - Spawning the server
  - Storing the loop data
  - Sending requests to the server
  - Sending replies to the client
  - Receiving server replies
  - Stopping the server
  - Much more not shown here: fault tolerance (supervised), redundancy, dynamic code update, statistics, logging, …
- Specific part
  - Server data:
    - Initialization
    - Handling
    - Cleanup
  - Format of client requests and replies
  - Basically, anything that is specific to a particular server

40

Callback modules

- The code is split up into two modules
  - Behavior module for the generic pattern
  - Callback module for the specific code
- The two modules agree on the function names and types in the callback API
- Advantages
  - Standardized programming style
  - Code reuse
  - Fewer bugs
  - Standardized abilities, such as logs, tracing, and statistics

Frequency server example



- Frequency allocator for cell phones
  - When a phone connects a call, it needs a frequency, which it asks from the frequency allocator
  - The client holds the frequency until the call is terminated, and then deallocates the frequency
- Clients and server are Erlang processes

Frequency server API

- Functional interface:

```
allocate() -> {ok,Frequency} | {error, no_frequency}
deallocate(Frequency) -> ok
```

- Start and termination:

```
start() -> true
stop() -> ok
```

- Frequency server has an internal set of frequencies, stored as a pair, free and allocated
  - Initially: free:{10,11,12,13,14,15}, alloc:{}
  - After allocating one frequency: free:{11,12,13,14,15}, alloc:{10}

43

Example execution

```
1> frequency:start().
true
2> frequency:allocate(), frequency:allocate(),
   frequency:allocate(), frequency:allocate(),
   frequency:allocate(), frequency:allocate().
{ok,15}
3> frequency:allocate().
{error,no_frequency}
4> frequency:deallocate(11).
ok
5> frequency:allocate().
{ok,11}
6> frequency:stop().
ok
```

Example server calls done from the Erlang interactive interface

The server has six available frequencies {10,11,12,13,14,15} and each allocate call will return one of them. The seventh call returns an error.

44

Frequency module

```
-module(frequency).
% External interface:
-export([start/0, stop/0,
         allocate/0, deallocate/1]).
% Callback interface:
-export([init/1, terminate/1 ,
         handle/2]).

start() ->
   server:start(frequency, []).
stop() ->
   server:stop(frequency).
allocate() ->
   server:call(frequency,
               {allocate, self()}).
deallocate(Freq) ->
   server:call(frequency,
               {deallocate, Freq}).
```

```
init(_Args) ->
   {[10,11,12,13,14,15], []}.

terminate(_Freqs) -> ok.

handle({allocate,Pid}, Freqs) ->
   allocate(Freqs, Pid);
handle({deallocate, Freq}, Freqs) ->
   {deallocate(Freqs, Freq), ok}.

allocate({[], Allocd}, _Pid) ->
   {{[],Allocd}, {error,no_freq}};
allocate({[Freq|Free], Allocd, Pid) ->
   {{Free, [{Freq,Pid}|Allocd]},
    {ok,Freq}}.

deallocate({Free,Allocd}, Freq) ->
   NewAllocd=
      lists:keydelete(Freq,1,Allocd),
   {[Freq|Free], NewAllocd}.
```


Generic server module

```
-module(server).
% Server interface:
-export([start/2, stop/1, call/2]).
-export([init/2]).

start(Name, Args) ->
   register(Name,
      spawn(server,init,[Name,Args])).

call(Name, Msg)→
   Name!{request,self(),Msg},
   receive
      {reply, Reply} -> Reply
   end.

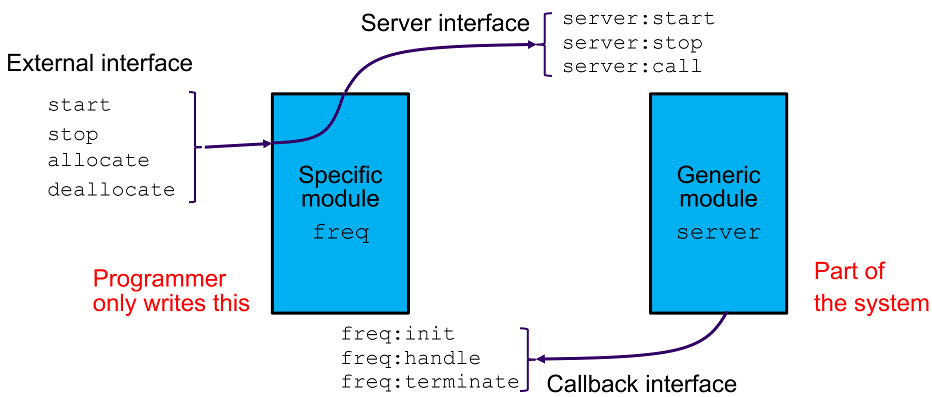
stop(Name) ->
   Name!{stop,self()},
   receive
      {reply,Reply} -> Reply
   end.
```

```
init(Mod, Args) ->
   State=Mod:init(Args),
   loop(Mod, State).

loop(Mod, State) ->
   receive
      {request, From, Msg} ->
         {NewState,Reply}=
            Mod:handle(Msg, State),
         reply(From, Reply),
         loop(Mod, NewState);
      {stop, From} ->
         Reply=Mod:terminate(State),
         reply(From, Reply)
   end.

reply(To, Reply) ->
   To!{reply,Reply}.
```

Architecture of generic server



47

Avoiding race conditions

48

More on message passing

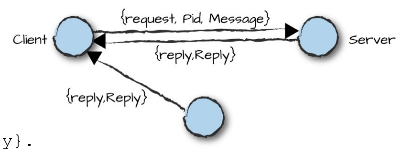
- Message passing has some hidden difficulties
- We show one of these difficulties and we explain how the gen_server handles it internally
  - The developer who uses gen_server does not have to handle these cases, they are done automatically

Race condition

- Assume we implement call/2 as follows:

```
call(Name, Message) ->
   Name ! {request, self(), Message},
   receive
      {reply, Reply} -> Reply
   end.

reply(Pid, Reply) -> Pid ! {reply, Reply}.
```



- How can we be confident that the reply is actually coming from the server and not from some other process?
  - With this implementation, we can't! But the solution to this problem is easy and can be done inside the gen_server.

Race condition solution

- To eliminate the race condition, we use a unique reference created by make_ref()
  - This ensures that the response is actually the reply to our message
  - This is very similar to the unique names in Oz returned by {NewName}
- The new call/2 is as follows:

```
call(Name, Msg) ->
   Ref=make_ref(),
   Name ! {request, {Ref, self()}, Msg},
   receive {reply, Ref, Reply} -> Reply end.

reply({Ref, To}, Reply) ->
   To ! {reply, ref, Reply}.
```

This is why receive can remove out of order

- This works because the receive statement removes the first message that matches the pattern (receive can remove messages in a different order than they arrived)

51

Synchronous and asynchronous calls

52

Synchronous calls

- A synchronous call waits for a reply
- This is what the server:call function does:

```
allocate() ->
   server:call(frequency, {allocate, self()}).

% Callback (in the client)
handle_call({allocate,Pid}, _From, Freqs) ->
   {NewFreqs, Reply} = allocate(Freqs, Pid),
   {reply, Reply, NewFreqs}.
```

53

Asynchronous calls

- Sometimes the client sends a message to the server but does not expect a reply
- The server:cast function handles that case
- The deallocate function in our frequency server example could use a cast, because it does not return any reply

```
deallocate(Freq) ->
   server:cast(frequency, {deallocate,Freq}).

% Callback (in the client)
handle_cast({deallocate,Freq}, Freqs) ->
   NewFreqs = deallocate(Freqs, Freq),
   {noreply, NewFreqs}.
```

54

Building a generic supervisor

55

Supervisor trees introduction

- In practical concurrent and distributed systems, it is observed that most faults and errors are transient
  - For example: network problems, timing problems, concurrent startup
  - Simple retrying is a surprisingly successful strategy
- Supervisor trees are designed to favor this strategy
  - Process sets are “supervised” (observed for failure) by supervisor processes
  - Supervisors have authority to stop and restart supervised processes
  - Supervisors are themselves observed, in case they fail
- Supervisor trees are carefully implemented to avoid races
  - Starting of a supervisor tree is synchronous to correctly initialize state

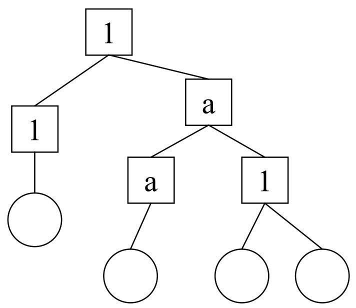
56

Supervisor structure and principles

- Supervisor tree is a hierarchy with a root
  - A supervisor tree consists of a set of supervisor nodes, organized as a hierarchy with a root node and internal nodes (the root is a very important process!)
  - A supervisor node is responsible for starting, stopping, and monitoring its child processes
  - All Erlang behaviors are designed to work together with supervisors
- Restart principles
  - Restart strategy: one_for_one, one_for_all, rest_for_one, simple_one_for_one
  - Restart frequency: the number of restarts is limited per time interval
    - If the limit is exceeded, the supervisor terminates and the next higher level supervisor takes some action
    - The intention is to prevent a situation where a process dies repeatedly for the same reason and is always restarted

57

Supervision hierarchies

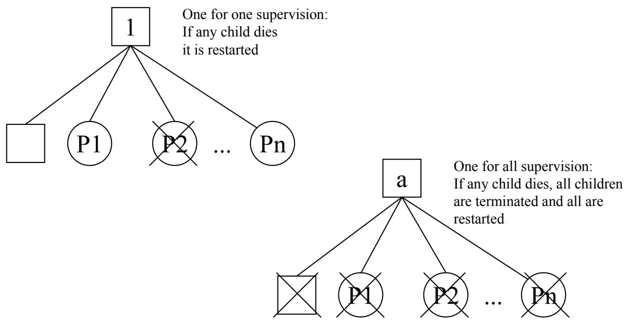


Abbreviations:
a One-for-all supervision
1 One-for-one supervision

- A supervisor (☐) is a process whose sole purpose is to start, monitor, and possibly restart workers
- A worker (○) is an instance of any behavior and it is managed by the supervisor

58

One-for-one and one-for-all supervision



- There are different forms of supervision depending on how the children processes work together:
  - One_for_one: if the children are independent (each manages one connection)
  - One_for_all: if the children are collaborating, then if one crashes they all have to be restarted, even the correct ones
  - Rest_for_one: all children started after the crashed process are terminated and restarted
  - Simple_one_for_one: all children are dynamic, started and stopped during application execution

A simple supervisor

- We show the code of a simple supervisor
  - If a child terminates abnormally, it is restarted
  - If a child terminates normally, it is removed from the supervision tree with no further action
  - Stopping the supervisor causes all the children to be terminated unconditionally
- This assumes that nothing abnormal happens when the supervisor starts the children! If a supervisor cannot start up normally, it aborts the startup procedure.
- We start each child by calling apply(Module,Function,Args) inside a list comprehension
- When the supervisor terminates, all linked processes receive an EXIT signal
- This example is taken from the excellent book *Designing for Scalability with Erlang/OTP* by Francesco Cesarini and Steve Vinoski

Simple supervisor code

```
-module(my_supervisor).
-export([start/2, init/1, stop/1]).

start(Name, ChildSpecList) ->
   register(Name,
     Pid=spawn(?MODULE, init,
               [ChildSpecList])),
   {ok,Pid}.

init(ChildSL) ->
   process_flag(trap_exit,true),
   loop(start_children(ChildSL)).

stop(Name) -> Name ! stop.

% Note list comprehension
start_children(ChildSpecList) ->
   [{element(2,apply(M,F,A)),{M,F,A}}
   || {M,F,A} <- ChildSpecList].
```

```
loop(ChildList) ->
   receive
      {'EXIT', Pid, normal} ->
         loop(lists:keydelete(Pid,
                     1,ChildList));
      {'EXIT', Pid, _Reason} ->
         NewChildList =
           restart_child(Pid,ChildList),
         loop(NewChildList);
      stop ->
         terminate(ChildList)
   end.

restart_child(Pid, ChildList) ->
   {Pid,{M,F,A}} =
      lists:keyfind(Pid,1,ChildList),
   {ok,NewPid} = apply(M,F,A),
   lists:keyreplace(Pid,1,
      ChildList,{NewPid,{M,F,A}}).

terminate(ChildList) ->
   lists:foreach(fun({Pid,_}) ->
      exit(Pid,kill) end, ChildList).
```

61

Some explanations

- ?MODULE is a macro whose value is the name of the current module
- The list comprehension loops over all tuples in ChildSpecList and creates the list ChildList
  - apply(M,F,A) calls the function F in module M with arguments A and returns the tuple {ok, Pid}
  - element(2,apply(M,F,A)) returns Pid, second element of the tuple
  - We assume that the function F spawns the child processes and links them to the parent
- The supervisor loops with a list of tuples ChildList
  - Each tuple is {Pid, {Module, Function, Argument}}
  - The second element of the tuple allows to restart the child

62

Towards a generic supervisor

- Generic part
  - Spawning the supervisor
  - Starting the children
  - Monitoring the children
  - Restarting the children
  - Stopping the supervisor
  - Cleaning up
- Specific part
  - What children to start
  - Specific child handling:
    - Start, restart
    - Child dependencies
  - Supervisor name
  - Supervisor behaviors

63

ETS tables for consistent restarting

- ETS tables are a stable storage, to allow restarting children in a consistent state
  - ETS is in-memory storage that is outside of the Erlang processes; when processes crash, ETS is unaffected
- An ETS table is linked to the process that creates it
  - If that process terminates, normally or abnormally, the ETS table is deleted
- With a supervisor tree, the ETS table is placed in a supervisor, not in a child process!
  - Pick the supervisor that monitors the processes using the table

64

Comparison with conventional approach

- The Erlang approach lets supervisors handle errors
  - When a child process has an error, it is designed to immediately crash, and the supervisor does the rest. The child does not handle its own errors!
  - All error conditions are mapped to one simple action: crash
- How does this compare to the conventional approach of defensive programming (lots of error checks in the children)?
- Several studies were done to compare the approaches
  - Heriot-Watt University made a study comparing Motorola's Data Mobility system (two-way radio communication streams) in C++ and Erlang
  - The Erlang implementation has 85% less code size. 27% of the C++ code consisted of error handling and defensive programming, versus only 1% of the Erlang code for the analogous operations.
  - Other studies give similar results. Ericsson has compared the MD110 corporate switch written in PLEX and in Erlang. They saw a tenfold decrease in code size.

65

Conclusions

66

Conclusions

- The Erlang language and OTP system are based on message passing between independent concurrent processes
  - We give a short refresher on Erlang's basic data types and message passing
  - We explain the primitives that Erlang uses to implement behaviors
- Erlang/OTP has support for behaviors, which are generic patterns for concurrent fault-tolerant systems
  - Generic server, finite state machine, event handler
  - Supervisor trees for fault tolerance ("Let it crash")
  - Stable storage (ETS, DETS, Mnesia) for consistent restart
- Erlang/OTP provides the primitives necessary to build highly resilient concurrent and distributed systems
  - Commercial applications prove the effectiveness of the Erlang approach

67

LINFO1131
Concurrent programming concepts

Lecture 9
Shared-state concurrency: introduction, locks, and tuple spaces

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Quick refresher on cells
- Large atomic actions
  - Why shared-state concurrency is difficult
  - The solution is to use large atomic actions
- Locks
  - The basic primitive for making a large atomic action
- Implementing locks
  - Simple locks and reentrant locks
- Tuple spaces
  - Another basic primitive: a multiset of tuples with send and receive
- Conclusions

2

Shared-state concurrency

- We have seen three good paradigms for concurrent programs
  - Deterministic dataflow: all the goodness of functional programming 😀 Good
  - Multi-agent programming: port objects and active objects, Erlang 😁 Better
  - Deterministic dataflow with ports: the best all-round paradigm 😍 Best!
- Now we will see a fourth paradigm, the worst of all!
  - Shared-state concurrency: using threads and cells together 👿
  - Three important concepts: locks, monitors, and transactions
  - **Locks** are important for accessing concurrent data structures
  - **Transactions** are important for database applications
  - **Monitors**, however, are deprecated and only recommended for legacy code
    - We will study them because they are still widely used (e.g., in Java)

3

Quick refresher on cells

4

Cells (mutable variables) (Section 6.3)

- Shared-state concurrency is based on the concept of mutable state, corresponding to "variables that can be assigned multiple times" in imperative programming languages such as Java and Python
- We this a cell to avoid confusion with the word "variable"
  - In mathematics, a variable in an expression is a placeholder for a value
  - In computing, a variable is an identifier, a variable in memory, or a cell
- A cell is a box with an identity and a content
  - The identity is a constant, called the "name" or "address" of the cell
  - The content is a variable in the single-assignment store

5

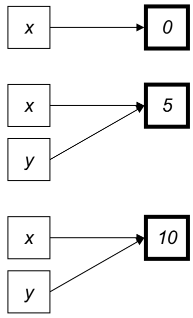
Cell operations

- We add three cell operations to the kernel language
- C={NewCell A}
  - Create a new cell with initial content A
  - Bind C to the cell's name
- C:=B (assign or write operation)
  - Replace the cell's content by B
- Z=@C (read or access operation)
  - Bind Z to the cell's content, without changing the content

6

Examples of using cells

- X={NewCell 0}
- X:=5
- Y=X
- Y:=10
- @X==10 % true
- X==Y % true



7

Cell semantics

- We extend the kernel language with three cell operations
- We extend the abstract machine to have two stores:
  - Variable store σ_1 (contains variables and their bindings)
  - Cell store σ_2 (contains cells, which are pairs of two variables)
- For example:
  - $\sigma_1 = \{x{=}\xi,\ y{=}\zeta,\ z,\ t{=}10,\ u{=}5,\ v,\ w\}$
  - $\sigma_2 = \{x{:}t,\ y{:}w\}$
- In σ_2 there are two cells, *x* and *y*
  - The name of *x* is the constant ξ, the name of *y* is ζ
  - The content of *x* is *t*, the content of *y* is *w*
- Assuming an environment {X→*x* ,Y→*y*,Z→*z*}:
  - The operation X:=Z changes *x*:*t* into *x*:*z*
  - The operation @Y returns the variable *w*

8

Exchange operation for concurrency

- The cell has a fourth operation called Exchange:
  - {Exchange C X Y}
    is the same as the two operations
  - X=@C C:=Y
    done atomically (i.e., it behaves as a single indivisible action)

- Exchange does a read and a write atomically
  - We will see that Exchange is very important for concurrent programs
  - For example, Exchange is essential to implement reentrant locks efficiently
  - Locks can be implemented without Exchange (it is possible with only atomic read and atomic write), but it is much, much more complicated (Dekker's algorithm)

- All processor architectures provide an instruction like Exchange
  - Many variations like "Test and Swap", "Test and Set", "Compare and Swap", but it always does atomic read and write

9

Large atomic actions

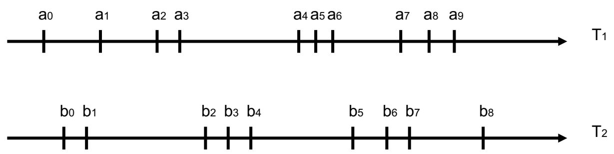
10

Shared-state concurrency (Chapter 8)

- Shared-state concurrency is defined as a programming paradigm where threads and cells are used together
- It is a widely used paradigm in industry today, and major languages (such as Java and C++) use this paradigm for concurrent programming
- Despite this popularity, it is the most difficult paradigm for concurrent programming
- We explain the reason for this difficulty and we give the main techniques for overcoming it

11

Why it is difficult (1)



- Consider two threads T_1 and T_2 that reference the cell C. Both threads do read and write operations on C.
- Assume that each thread does n operations on C. The scheduler can choose $\binom{2n}{n}$ possible interleavings, which is equal to $(2n)!/(n!)^2$
  - There are 2n operation instants in all, T_1 picks n of these, and T_2 gets what's left
  - Using Stirling's formula $n! \approx (n/e)^n \sqrt{2\pi n}$ we get $2^{2n}/\sqrt{\pi n}$, which is exponential in n

12

Why it is difficult (2)

- The number of possible interleavings is an exponential function of the number of operations
  - If the program is correct, then all interleavings must be correct
  - Even a very small number of incorrect interleavings will quickly break the system, given the high speed of computers
  - Scheduler timing cannot be relied upon to improve things, because of the program's execution environment
  - Testing can only check a small fraction of the interleavings
  - Verification (manual by proofs, or automatic by model checking) is also strongly limited because it requires massive computation, so it is limited to verifying only part of a large program

Testing and verification are still essential! It's just that they are not enough by themselves.

- Therefore, the only way to write correct concurrent programs is to make them correct by design

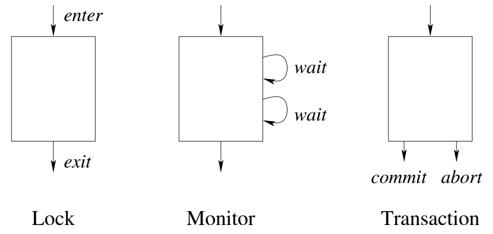
13

Designing correct concurrent programs

- Designing correct concurrent programs means to manage the interleavings
  - We use a technique that reduces the number of interleavings to a much smaller number, which can be checked
- There are three ways to do this
  - Deterministic dataflow with ports: restrict the paradigm so that all interleavings give the same result, except for a small number of ports for which all interleavings must be checked
    - Use deterministic dataflow most of the time, add a small number of ports only where they are really needed (typically: interaction with the real world)
  - Message passing between port objects: internally, each port object executes in a single thread, so that we can reason about interleavings between method executions and not between single operations
    - Used successfully by Erlang and other multi-agent languages
  - This lecture
  - Atomic actions on shared cells: we group thread operations together into large atomic actions, so that the number of interleavings is much reduced

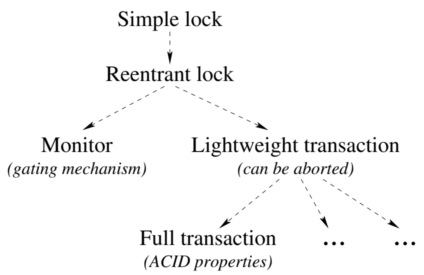
14

Programming with atomic actions



- The main technique for building correct shared-state concurrent programs is to build large atomic actions
- The basic concept is the lock, which makes a group of operations atomic
- This concept has two important refinements, the monitor and the transaction

Hierarchy of atomic actions



- The atomic actions can be put in a hierarchy. Monitors and transactions are locks with added functionality.

Reasoning with atomic actions

- Consider a program that uses atomic actions throughout
  - Well-chosen atomic actions will often coincide with abstraction boundaries, so that correctness proof of the concurrent execution is very similar to the correctness proof of the abstractions themselves.
- Proving correctness then consists of two parts:
  - Proving that each atomic action is correct by itself (safety): assume that there is an invariant assertion and show that the invariant assertion is preserved by each atomic action.
  - Proving that the sequence of atomic actions makes progress (liveness): this means to show that the program using the atomic actions makes progress toward its goal. This requires an assertion that measures progress.
    - For example, in the lift control system: that the lift will always eventually go to the lift that is first in the schedule and correctly update the schedule

17

Locks

18

Locks (Section 8.3)

- A lock is a language concept that guarantees that only one thread will be running in a specific part of the program.
  - This part of the program is called a critical section
- If another thread tries to enter the critical section, it will wait at the boundary until there are no threads inside

- A critical section guarded by lock L does not have to be contiguous, for example:

```
   …
   …
   …
   …
  […
L […
  […
   …
   …
  […
L […
  […
```

Lock L protects a critical section that consists of two different parts of the program code

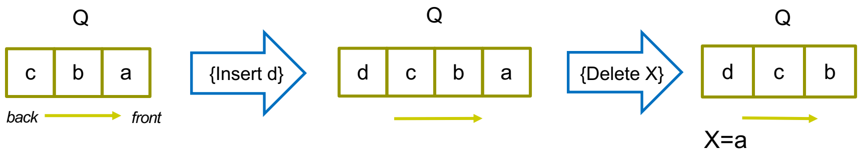
19

Lock abstraction

- We define a lock abstraction with the following operations:
  - L={NewLock}: returns a new lock L
  - {IsLock L}: returns true if and only if L is a lock, otherwise false
  - **lock** L **then** <stmt> **end**: guards a statement with lock L
- If no thread is in any statement guarded by a lock, then any thread can enter. A thread waits (suspends) if it attempts to enter a guarded statement while there is another thread inside.
- If a thread is currently executing a guarded statement, then the same thread can enter again (the lock statement can be nested). This property is called reentrancy.
- The lock statement can be called in different parts of the program with the same lock. The lock will ensure that at most one thread is inside any of the parts that it guards.

20

Concurrent queue abstraction



- Let us define a concurrent queue abstraction using cells and locks
- A queue is a sequence of elements with insert and delete operations
  - {Insert X} adds X to the back of the queue
  - {Delete X} removes an element from the front of the queue and binds it to X

21

Queue internal data representation

Q=q(3 a|b|c|B B) {Insert d} Q=q(4 a|b|c|d|B2 B2) {Delete X} Q=q(3 b|c|d|B2 B2)
X=a

- We implement the queue as a 3-tuple q(N F B) where N is the number of elements and F is a difference list with tail B
  - For example, q(3 F B) with F=a|b|c|B
  - Insertion is done by binding B
  - Deletion is done by traversing F
- The queue is stored in a cell so we can update it
  - The tail must always be directly accessible
  - We use a lock so that we can do both a read and write of the cell

22

Queue code with locks

```
fun {NewQueue}
   X C={NewCell q(0 X X)}
   L={NewLock}
   proc {Insert X}
      …
   end
   proc {Delete X}
      …
   end
in
   q(insert:Insert delete:Delete)
end
```

```
proc {Insert X}
N F B2 in
   lock L then
      q(N F X|B2)=@C
      C:=q(N+1 F B2)
   end
end
```

```
proc {Delete X}
N F2 B in
   lock L then
      q(N X|F2 B)=@C
      C:=q(N-1 F2 B)
   end
end
```

23

Why we need reentrancy

- Let's extend our queue abstraction with a new operation to insert two elements together (with nothing in between):

```
proc {Insert2 X Y}
   {Insert X}
   {Insert Y}
end
```

- How can we guarantee that no other thread will come in between?

```
proc {Insert2 X Y}
   lock L then
      {Insert X}
      {Insert Y}
   end
end
```

When we call {Insert X} the thread is already inside the lock. This means the lock has to be reentrant!

24

The advantage of single assignment

- Our queue implementation has the very nice property that it is possible to delete elements before they are inserted!
  - If the queue is empty, then {Delete X} will return unbound X and the queue has -1 elements (!)
  - Doing {Insert aaa} after that will bind X to aaa and the queue will have 0 elements
- You can verify this by running {Delete X} on an empty queue and seeing what happens to F and B
  - It works because the variable store contains logical formulas (variable bindings are actually logical equalities)
  - Variable binding is adding logical information to the store; the system is doing logic programming
- If you want the queue to block when trying to delete from an empty queue, then you have to call Wait explicitly:
  - {Delete X} {Wait X}

25

Queue code with Exchange

- We can implement the queue using the cell Exchange operation:

```
        {Exchange C X Y}
is the same as:
        X=@C  C:=Y   % Atomic
```

```
fun {NewQueue}
   X C={NewCell q(0 X X)}
   proc {Insert X}
      …
   end
   proc {Delete X}
      …
   end
in
   queue(insert:Insert delete:Delete)
end
```

```
proc {Insert X}
N F B2 M in
   {Exchange C q(N F X|B2) q(M F B2)}
   M=N+1
end

proc{Delete X}
N F2 B M in
   {Exchange C q(N X|F2 B) q(M F2 B)}
   M=N-1
end
```

We still need locks if we do more than one read and one write operation (for example, Insert2 still needs a lock)!

26

Implementing locks

27

Implementing locks

- We will implement locks using cells and Exchange
- We will do it in three steps:
  - A simple lock (non-reentrant version)
  - A simple lock with exception handling
  - A reentrant lock with exception handling

Token passing is a powerful technique that is used often in concurrent and distributed programming

- We use a technique called token passing
  - To enter a critical section, a thread needs to possess a token
  - There is only one token in the system, so only one thread can be inside
  - If a thread tries to get in without a token, then it waits for the token
  - When a thread leaves the critical section, it forwards the token to the next thread

28

Simple lock

- The simple lock does only token passing:

```
fun {SimpleLock}
   Token={NewCell unit}
   proc {Lock P}
   Old New in
      {Exchange Token Old New}
      {Wait Old} % Enter the critical section (get the token)
      {P}
      New=unit  % Leave the critical section (forward the token)
   end
in
   'lock'('lock':Lock)
end
```

This uses higher-order programming: P is a zero-argument procedure that contains a statement: P = **proc** {$} <stmt> **end**

29

Simple lock with exception handling

- What happens if the <stmt> raises an exception?
  We still need to pass the token!

```
fun {SimpleLock}
   Token={NewCell unit}
   proc {Lock P}
   Old New in
      {Exchange Token Old New}
      {Wait Old}
      try {P} finally New=unit end
   end
in
   'lock'('lock':Lock)
end
```

This passes the token to the next thread even if {P} raises an exception

30

Reentrant lock

- A reentrant lock needs to know which thread is inside the lock
  - It needs a new concept: the thread's identity, which in Oz is returned by {Thread.this}
```
fun {NewLock}
   Token={NewCell unit}
   CurThr={NewCell unit}
   proc {Lock P}
     …
   end
in
   'lock'('lock':Lock)
end
```

```
proc {Lock P}
   if {Thread.this} == @CurThr then
        {P}
   else Old New in
        {Exchange Token Old New}
        {Wait Old}
        CurThr:={Thread.this}
        try {P} finally
           CurThr:=unit
           New=unit
        end
   end
end
```

What can go wrong if these two statements are switched?

Debugging the lock implementation

- What can go wrong if these instructions are switched?
  {Wait Old}
  CurThr:={Thread.this}
- What can go wrong if these instructions are switched?
  CurThr:=unit
  New=unit
- Hint:

```
           {Wait Old}            <- Enter critical section
Critical [ CurThr:={Thread.this}
section  [ {P}
         [ CurThr:=unit
           New=unit              <- Exit critical section
```

Imagine what can happen if CurThr is changed outside of the critical section

Tuple spaces: a concept in between sharing and messages

33

Tuple spaces

- A tuple space TS is a multiset of tuples with three operations:
  - {TS write(T)} : add the tuple T to TS
  - {TS read(L T)} : wait until the tuple space contains at least one tuple with label L, then remove one such tuple and bind it to T
  - {TS readnonblock(L T B)} : same as read, but returns immediately. Binds B=false if the tuple space contains no tuple with label L. Otherwise binds B=true, removes one such tuple and binds it to T.
  - (The original tuple space concept invented by David Gelernter in 1985, called Linda, had a more general read operation that does pattern matching.)
- The tuple space abstraction is sometimes called a coordination model
  - A language with a tuple space is called a coordination language
  - Modern publish/subscribe systems are examples of coordination models

34

Combining shared state and message passing

- The tuple space concept is a combination of shared state and message passing
- Like shared state because the tuple space is shared between threads
  - It is easier to program than shared state because the write and read operations are like large atomic actions, i.e., they add a large chunk of information to the tuple space
- Like message passing because tuples are sent and received:
  - Writing a tuple is like a send and reading a tuple is like a receive
  - The difference with message passing is the sender's knowledge of the receiver: with message passing, the sender knows the receiver, whereas with tuple spaces, the sender does not know the receiver

35

Queue code with tuple spaces

- A concurrent queue can easily be implemented using tuple spaces
- The tuple space implementation uses one read and one write for a queue operation, but it does not need a lock because tuples are unique entities, like tokens. Reading a tuple behaves like entering a critical section: the first thread can do it immediately, the others wait.
- It is interesting to compare the three implementations:
  - Queue with lock: insert with three operations (lock, read, write)
  - Queue with Exchange: insert with one operation, no locks needed
  - Queue with tuple space: insert with two operations, no locks needed

36

Queue code with tuple spaces

- ```
  fun {NewQueue}
     X TS={New TupleSpace init}
     proc {Insert X}
        …
     end
     proc {Delete X}
        …
     end
  in
     {TS write(q(0 X X))}
     queue(insert:Insert
           delete:Delete)
  end
  ```

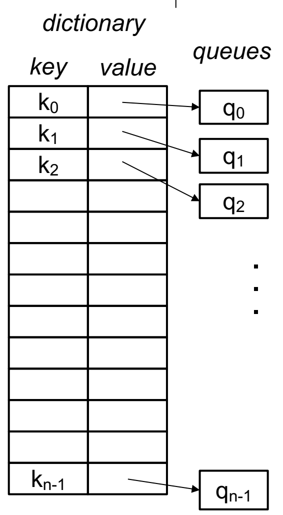
```
proc {Insert X}
N F B2 in
   {TS read(q q(N F X|B2))}
   {TS write(q(N+1 F B2))}
end
```

```
proc {Delete X}
N F2 B in
   {TS read(q q(N X|F2 B))}
   {TS write(q(N-1 F2 B))}
end
```

37

Implementing tuple spaces

- We implement the tuple space abstraction with a lock, a dictionary, and a concurrent queue
  - Note that that the inverse is also possible: a concurrent queue can also be implemented with a tuple space
- A dictionary is a form of dynamic hash table:
  - A dictionary is a dynamic set of key/value pairs, where pairs can be added, removed, and tested for presence, in constant time
  - In the tuple space, the keys are tuple labels and the values are queues (for each label, there is a queue of tuples of that label)
- The tuple operations will insert and delete elements in the queue stored at the label's position in the dictionary



38
```

Conclusions

39

Conclusions

- Shared-state concurrency is a programming paradigm that combines threads and cells (mutable state)
  - It is difficult because the number of interleavings is exponential in the number of operations on shared data
- Concurrent programs are designed by building large atomic actions to reduce the number of interleavings
  - Three main kinds of atomic actions are locks, monitors, and transactions
  - Tuple spaces are a combination of message passing and shared state
- Today we showed how to use locks and how to implement them
- In the next lectures we will focus on monitors and transactions

40

```
% LINFO1131
% Advanced Programming Language Concepts

% Lecture 9 (Nov. 29, 2023)

% Introduction to shared-state concurrency
% - Concurrent queue, locks
% - Tuple spaces

%%%%%%%%%%%%%%%%%%%%%%%

% 1. Mutable state (cells)

declare
C={NewCell 0}1

{Browse @C}

{Browse C}

C := @C + 1

{Browse @C}

declare
D=C

{Browse @D}

declare
E={NewCell 100}

{Browse @E}

{Browse C==D}

{Browse C==E}

%%%%%%%%%%%%%%%%%%%%%%%

% 2. Concurrent queue
% Example of a concurrent abstraction defined with locks
declare
fun {NewQueue}
   X C={NewCell q(0 X X)}
   L={NewLock}
   proc {Insert X}
      N F B2
   in
      lock L then
         q(N F X|B2)=@C
         C:=q(N+1 F B2)
      end
   end
```

```
   proc {Delete X}
      N F2 B
   in
      lock L then
         q(N X|F2 B)=@C
         C:=q(N-1 F2 B)
      end
   end
in
   q(insert:Insert delete:Delete)
end

declare
Q={NewQueue}

{Q.insert a}

local X in {Q.delete X} {Browse X} end

%%%%%%%%%%%%%%%%%%%%%%%

% 3. Implementation of simple lock
% Simple locks use token passing to enforce mutual exclusion.
% See the slides to see how this is extended to reentrant locks.
declare
fun {SimpleLock}
   Token={NewCell unit}
   proc {Lock P}
      Old New
   in
      {Exchange Token Old New} % Get a place in line
      {Wait Old} % Wait until previous thread gives me token
      {P}  % Inside of the critical section
      New=unit % Give the token to the next thread
   end
in
   'lock'('lock':Lock)
end

% Create a lock and use it
declare D L in
D={NewCell 10}
L={SimpleLock}.'lock'
thread {L proc {$} D:= @D +1 end} end  % Two operations!
thread {L proc {$} D:= @D +1 end} end
% With no locks, this code is buggy!
% Scheduler might stop just after @D operations, so final
% result is 11 and not 12 (with no locks)
% With lock, result will always be 12.

{Browse @D}

%%%%%%%%%%%%%%%%%%%%%%%
```

```
% 3bis. Reentrant lock
% This definition returns the lock directly in the argument L.
% This definition also works correctly if {P} raises an exception.
declare
proc {ReentrantLock L}
   Token={NewCell ok}
   CurThr={NewCell none}
in
   proc {L P}
      if {Thread.this}==@CurThr then
         {P}
      else
         Xold Xnew
      in
         {Exchange Token Xold Xnew}
         {Wait Xold}
         CurThr:={Thread.this}
         try
            {P}
         finally
            CurThr:=none
            Xnew=ok
         end
      end
   end
end

%%%%%%%%%%%%%%%%%%%%%%%

% 4. Tuple spaces

% 4.1 Queue abstraction for tuple space
% This is the queue used to implement tuple spaces

declare
fun {NewQueue}
   X in
   q(0 X X)
end

fun {Insert q(N S E) X}
   E1 in
   E=X|E1 q(N+1 S E1)
end

fun {Delete q(N S E) X}
   S1 in
   S=X|S1 q(N-1 S1 E)
end

fun {DeleteNonBlock q(N S E) X}
   if N>0 then H S1 in
      X=[H] S=H|S1 q(N-1 S1 E)
   else
```

```
      X=nil q(N S E)
   end
end

fun {DeleteAll q(_ S E) L}
   X in1
   L=S E=nil
   q(0 X X)
end

fun {Size q(N _ _)} N end

%%%%%%%%

% 4.2 Tuple space implementation

declare
class TupleSpace
   prop locking
   attr tupledict

   meth init tupledict:={NewDictionary} end

   meth EnsurePresent(L)
      if {Not {Dictionary.member @tupledict L}}
      then @tupledict.L:={NewQueue} end
   end

   meth Cleanup(Q L)
      @tupledict.L:=Q
      if {Size Q}==0
      then {Dictionary.remove @tupledict L} end
   end

   meth write(Tuple)
      lock L={Label Tuple} in
         {self EnsurePresent(L)}
         @tupledict.L:={Insert @tupledict.L Tuple}
      end
   end

   meth read(L Tuple) X in
      lock Q in
         {self EnsurePresent(L)}
         Q={Delete @tupledict.L X}
         {self Cleanup(Q L)}
      end
      {Wait X} X=Tuple
   end

   meth readnonblock(L Tuple ?B)
      lock U Q in
         {self EnsurePresent(L)}
         Q={DeleteNonBlock @tupledict.L U}
```

```
         case U of [X] then
            {self Cleanup(Q L)} B=true X=Tuple
         else B=false end
      end
   end
end

%%%%%%%%

% 4.3 Tuple space examples

declare
TS={New TupleSpace init}

{TS write(foo(1))}
{TS write(foo(1 2))}
{TS write(bar(2))}

% Waits until a tuple with label 'foo' is in TS
local X in {TS read(foo X)} {Browse X} end

local X in {TS read(bar X)} {Browse X} end
{TS write(car(5))}
{TS write(car(6))}

{TS write(foo(a))}

%%%%%%%%%%%%%%%%%%%%%%%
```

LINFO1131
Concurrent programming concepts

Lecture 10
Shared-state concurrency: monitors

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Monitor definition
  - Intuition: why do we need monitors?
  - Semantics: wait, notify, notifyAll
- Bounded buffer
  - Definition
  - Implementation, how to use monitor
  - Buggy version: why is it buggy?
- Programming pattern
  - To guarantee bug-free use of monitors
- Implementation
  - Get-release lock
  - Queue
  - Code for wait, notify, notifyAll

2

Monitor definition

3

Monitors (Section 8.4)

- Monitors are an extension of locks for coordinating threads when they interact with data abstractions
- Locks alone are not sufficient
  - Consider a bounded buffer. It is not enough to protect the buffer with a lock. What happens if a thread wants to put an element in the buffer, and the buffer is full? What happens if a thread wants to remove an element from the buffer, and the buffer is empty? We need a way for the thread to wait until the buffer is nonfull or nonempty. This cannot be done with just locks.
- Monitors were introduced by Per Brinch Hansen in 1972 and developed by C. A. R. (Tony) Hoare in 1974
  - They are still widely used today, for example, synchronized objects in Java are a form of monitors

4

Monitor intuition

- A monitor adds two operations, wait and notify, and a wait set
  - The wait set consists of a set of suspended threads
  - Wait and notify are used to manage how threads enter and exit the wait set
  - Wait and notify are only possible when inside the lock
- Wait operation: when a thread calls wait, it suspends, is put in the wait set, and the monitor lock is released
- Notify operation: when a thread calls notify, it wakes one thread in the wait set. The woken thread tries to get the monitor lock again, at the place where it was suspended (i.e., the wait operation continues)

Monitor semantics

- We give the Java semantics, because it is simple and popular
  - In Java, a monitor is an object extended with an internal lock and a wait set
  - There are three operations, wait, notify, and notifyAll
- Wait operation is defined as follows:
  - Suspend the current thread; place the thread in the wait set; release the lock
  - When the wait returns, the thread tries to get the lock again
- Notify operation is defined as follows:
  - If the wait set is nonempty, remove an arbitrary thread T
  - Resume execution of T at the point it was suspended (i.e., at its call to wait).
  - T proceeds to get the lock like any other thread. (Note that T always suspends again briefly, until the notifying thread releases the lock.)

NotifyAll operation

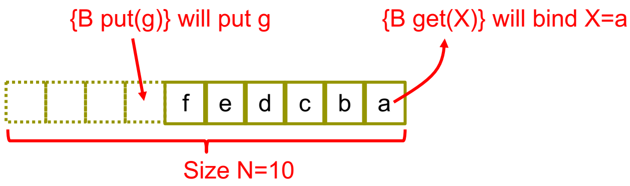
- The notify operation resumes an arbitrary thread
  - This can be a problem if there are multiple threads in the wait set: how does notify know which one to resume?
- The notifyAll operation does notify for all threads in the wait set
  - All threads are resumed; all threads try to get the lock
  - The wait set is emptied
- One thread will get the lock and execute the monitor. If it is the wrong thread, it will do a wait and be put back into the wait set. If it is the right thread, it will do the desired operation.
  - This can lead to contention if there are many threads. If this is a problem, there are ways to avoid it, for example by using multiple wait sets.

7

Bounded buffer

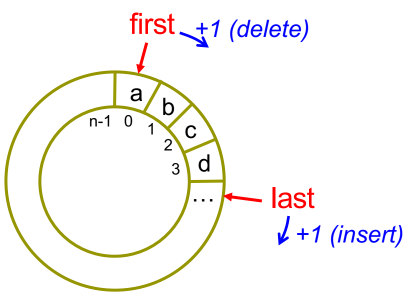
8

Bounded buffer



- We define a concurrent bounded buffer using a monitor
- A bounded buffer is a concurrent queue data abstraction with fixed maximum size. It has three operations:
  - B={New Buffer init(N)} : create a new buffer of size N
  - {B put(X)} : put element X in the buffer. If the buffer is full, wait until it has room.
  - {B get(X)} : remove an element from the buffer and bind it to X. If the buffer is empty, wait until it contains at least one element

Bounded buffer implementation



- The bounded buffer can be efficiently implemented as an array of size *n* organized as a ring (with wraparound)
  - Elements are removed by incrementing first
  - Elements are inserted by incrementing last
  - If an index is outside [*0*,*n-1*], it is replaced by the index modulo *n*
  - If the number of elements last–first would become <0 or >n-1, the operation is blocked

Bounded buffer code without monitor

```
class Buffer
   attr buf first last n i
   meth init(N)
         buf:={NewArray 0 N-1 null}
         first:=0 last:=0 n:=N i:=0
   end
   meth put(X)
         …
   end
   meth get(X)
         …
   end
end
```

```
meth put(X)
   … % wait until i<n
   % now add an element:
   @buf.@last:=X
   last:=(@last+1) mod @n
   i:=@i+1
end
```

```
meth get(X)
   … % wait until i>0
   % now remove an element:
   X=@buf.@first
   first:=(@first+1) mod @n
   i:=@i-1
end
```


Monitor abstraction

- We define the monitor abstraction as follows:
- M={NewMonitor}: create a new monitor as a record:

  M=monitor('lock':LockM
  wait:WaitM notify:NotifyM notifyAll:NotifyAllM)

- With the following four operations:
  - {M.'lock' **proc** {$} <stmt> **end**}: reentrant lock
  - {M.wait}: wait operation
  - {M.notify}: notify operation
  - {M.notifyAll}: notifyAll operation

Bounded buffer code complete

```
class Buffer
    attr m buf first last n i
    meth init(N)
            m:={NewMonitor}
            buf:={NewArray 0 N-1 null}
            first:=0 last:=0 n:=N i:=0
    end
    meth put(X)
            …
    end
    meth get(X)
            …
    end
end
```

```
meth put(X)
   {@m. 'lock' proc {$}
           if @i>=@n then % if full, wait
              {@m.wait}
              {self put(X)} % try again!
           else
              @buf.@last:=X
              last:=(@last+1) mod @n
              i:=@i+1
              {@m.notifyAll} % tell others!
           end
   end}
end
```

13

Bounded buffer code buggy version!

```
class Buffer
    attr m buf first last n i
    meth init(N)
            m:={NewMonitor}
            buf:={NewArray 0 N-1 null}
            first:=0 last:=0 n:=N i:=0
    end
    meth put(X)
            …
    end
    meth get(X)
            …
    end
end
```

This version is buggy!
Find a scheduler scenario that makes it go wrong.

```
meth put(X)
   {@m. 'lock' proc {$}
           if @i>=@n then % if full, wait
              {@m.wait}
           end
           @buf.@last:=X
           last:=(@last+1) mod @n
           i:=@i+1
           {@m.notifyAll}
   end}
end
```

14

Monitor programming pattern

15

Monitor programming pattern

- The technique used in the bounded buffer can be used elsewhere
- We define a general programming pattern for monitors:

```
meth methHead
   lock
        while not <expr> do wait;
        <stmt>
        notifyAll;
   end
end
```

- When the wait returns it always asks for the lock again, just like it was going in the critical section the first time

- We can use tail recursion instead:
  - They have the same semantics!
  - Oz does not have while loops (strangely!)

```
meth methHead
   lock
         if not <expr> then
            wait; {self methHead}
         else
            <stmt>
            notifyAll;
         end
   end
end
```

16

Implementing monitors

17

Implementing monitors

- We will implement monitors as an extension of reentrant locks
- Compared to the reentrant lock implementation we need to make two changes:
  - We modify the lock to be a get-release lock, so that entering and exiting the critical section are separate operations, called get and release
  - We add a queue to implement the wait set. Implementing it as a queue gives good behavior because it ensures fairness: it's not possible for a thread to stay forever in the queue while other threads are leaving the queue.
    - The queue has three extra operations: DeleteAll, Size, and DeleteNonBlock (which returns [X] if the queue is nonempty and nil if it is empty)

18

Get-release lock

```
fun {NewGRLock}
   Token1={NewCell unit}
   Token2={NewCell unit}
   CurThr={NewCell unit}

   fun {GetLock}
         …
   end
   proc {ReleaseLock}
         CurThr:=unit
         unit=@Token2  % Pass the token
   end
in
   'lock'(get:GetLock
          release:ReleaseLock)
end
```

```
fun {GetLock}
   if {Thread.this}\=@CurThr then
         Old New
   in
         {Exchange Token1 Old New}
         {Wait Old}
         Token2:=New  % Prepare release
         CurThr:={Thread.this}
         true
   else
         false
   end
end
```

19

Extended queue

- This extends the queue we did before with a cell and a lock

```
fun {NewQueue}
   X C={NewCell q(0 X X)}
   L={NewLock}
   proc {Insert X}
         …  % As before
   end

   proc {Delete X}
         …  % As before
   end

   fun {Size}
         lock L then @C.1 end
   end
```

```
   fun {DeleteAll}
         lock L then X S E in
            q(_ S E)=@C
            C:=q(0 X X)   % Make empty
            E=nil  S   % Return all
         end
   end
   fun {DeleteNonBlock}
         lock L then
            if {Size}>0 then [{Delete}]
            else nil end
         end
   end
in
   queue(insert:Insert delete:Delete size:Size
         deleteall:DeleteAll
         deleteNonBlock:DeleteNonBlock)
end
```

20

Monitor code

```
fun {NewMonitor}
   Q={NewQueue}
   L={NewGRLock}

   proc {LockM P}
         if {L.get} then
            try {P} finally {L.release} end
         else {P} end
   end

   proc {WaitM}
   X in
         {Q.insert X} {L.release}
         {Wait X} if {L.get} then skip end
   end

   proc {NotifyM}
   U={Q.deleteNonBlock} in
         case U of [X] then X=unit
         else skip end
   end

   proc {NotifyAllM}
   L={Q.deleteAll} in
         for X in L do X=unit end
   end
in
   monitor('lock':LockM wait:WaitM
           notify:NotifyM
           notifyAll:NotifyAllM)
end
```

21

Conclusions

22

Conclusions

- Monitors extend locks with the ability to suspend and resume threads depending on conditions specific to the data abstraction
  - Wait set: set (or queue) of suspended threads
  - Wait and notify operations: add/remove one thread in the wait set
  - NotifyAll operation: removing all threads, almost always the correct operation
- Monitors are difficult to program with, unless you use a pattern
  - We have shown a general pattern for programming with monitors
  - Monitors are widely used in legacy code, but we do not recommend them for new code! They should be deprecated everywhere!
- In the next lecture we will see another major extension of locks, namely transactions, which are key operations for large databases

```
% LINFO1131
% Advanced Programming Language Concepts

% Lecture 10 (Dec. 6, 2023)

% Monitor implementation

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Queue data structure
% Used to implement wait set

declare
fun {NewQueue}
   X in
   q(0 X X)
end

fun {Insert q(N S E) X}
   E1 in
   E=X|E1 q(N+1 S E1)
end

fun {Delete q(N S E) X}
   S1 in
   S=X|S1 q(N-1 S1 E)
end

fun {DeleteNonBlock q(N S E) X}
   if N>0 then H S1 in
      X=[H] S=H|S1 q(N-1 S1 E)
   else
      X=nil q(N S E)
   end
end

fun {DeleteAll q(_ S E) L}
   X in
   L=S E=nil
   q(0 X X)
end

fun {Size q(N _ _)} N end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Correct implementation of monitors
% Combination of reentrant lock and queue
% Reentrant lock is split into two operations: get and release
% Queue is used as wait set for threads: a thread waits
% by means of a dataflow variable

% Book version may be incorrect (correct in 4th & later printings)!
% Code below includes bug fix (see book Errata page)
```

```
declare
proc {NewMonitor ?LockM ?WaitM ?NotifyM ?NotifyAllM}
   Q={NewCell {NewQueue}}
   Token1={NewCell unit}
   Token2={NewCell unit}
   CurThr={NewCell unit}

   % Returns true if got the lock, false if not (already inside)
   fun {GetLock}
      if {Thread.this}\=@CurThr then Old New in
         {Exchange Token1 Old New}
         {Wait Old}
         Token2:=New
         CurThr:={Thread.this}
         true
      else false end
   end

   proc {ReleaseLock}
      CurThr:=unit
      unit=@Token2
   end
in
   proc {LockM P}
      if {GetLock} then
         try {P} finally {ReleaseLock} end
      else {P} end
   end

   proc {WaitM}
   X in
      Q:={Insert @Q X}
      {ReleaseLock} {Wait X}
      if {GetLock} then skip end
   end

   proc {NotifyM}
   X in
      Q:={DeleteNonBlock @Q X}
      case X of [U] then U=unit else skip end
   end

   proc {NotifyAllM}
   L in
      Q:={DeleteAll @Q L}
      {ForAll L proc {$ X} X=unit end}
   end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% LINFO1131
% Advanced Programming Language Concepts

% Lecture 10 (Dec. 6, 2023)

% Shared-state concurrency
% - Bounded buffer with monitors

%%%%%%%%%%%%%%%%%%%

% 1. Bounded buffer (Buggy version)
declare
class Buffer
   attr
      buf first last n i
      lockm waitm notifym notifyallm

   meth init(N)
      buf:={NewArray 0 N-1 null}
      first:=0 last:=0 n:=N i:=0
      {NewMonitor @lockm @waitm @notifym @notifyallm}
   end

   meth put(X)
      {@lockm
       proc {$}
          % Wait until buffer is not full (@i<@n)
          % BUGGY because other thread can slip in
          if @i==@n then {@waitm} end
          % Now add one element:
          @buf.@last:=X
          last:=(@last+1) mod @n
          i:=@i+1
          {@notifyallm}
      end}
   end

   meth get(X)
      {@lockm
       proc {$}
          % Wait until buffer is not empty (@i>0)
          if @i==0 then {@waitm} end
          % Now remove one element:
          X=@buf.@first
          first:=(@first+1) mod @n
          i:=@i-1
          {@notifyallm}
       end}
   end
end

%%%%%%%%%%%%%%%%%%%

% 2. Bounded buffer (correct version)
```

```
declare
class Buffer
   attr
      buf first last n i
      lockm waitm notifym notifyallm

   meth init(N)
      buf:={NewArray 0 N-1 null}
      first:=0 last:=0 n:=N i:=0
      {NewMonitor @lockm @waitm @notifym @notifyallm}
   end

   meth put(X) /* correct version */
      {@lockm
       proc {$}
          % Wait until buffer is not full (@i<@n)
          if @i==@n then
             {@waitm}
             /* condition might become false here */
             {self put(X)} /* test cond. again */
          else
             % Now add one element:
             @buf.@last:=X
             last:=(@last+1) mod @n
             i:=@i+1
             {@notifyallm}
          end
      end}
   end

   meth get(X)
      {@lockm
       proc {$}
          % Wait until buffer is not empty (@i>0)
          if @i==0 then
             {@waitm}
             /* condition might become false here */
             {self get(X)} /* test condition again */
          else
             % Now remove one element:
             X=@buf.@first
             first:=(@first+1) mod @n
             i:=@i-1
             {@notifyallm}
          end
       end}
   end
end

%%%%%%%%%%%%%%%%%%%

% 3. Example execution of bounded buffer

declare
```

```
BB={New Buffer init(3)}

{BB put(a)}

local X in {BB get(X)} {Browse X} end

{BB put(a)}
{BB put(b)}
{BB put(c)}
{Browse 'try fourth'}
{BB put(d)}
{Browse 'end fourth'}

local X in {BB get(X)} {Browse X} end

local X in {BB get(X)} {Browse X} end
{Browse 'after get'}

local X in {BB get(X)} {Browse X} end
{Browse 'after get'}

{BB put(f)}

%%%%%%%%%%%%%%%%%%%
```

LINFO1131
Concurrent programming concepts

Lecture 11
Shared-state concurrency: transactions

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Concurrency control definition
- Safety and liveness
- Concurrency control concepts
  - Locks for safety
  - Timestamps for liveness
  - Optimism versus pessimism
- A simple transaction system
  - A naïve transaction manager
  - Deadlocks
  - A correct transaction manager
- Implementation of the transaction system
- Conclusions

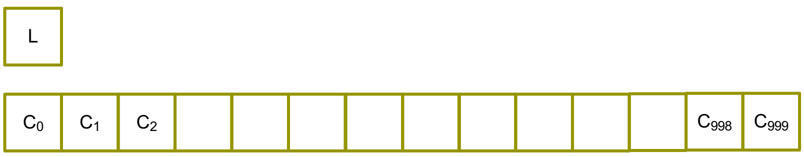
2

Transactions (Section 8.5)

- Transactions were introduced as a basic concept for management of large databases that must sustain a high rate of concurrent updates while keeping data coherent and surviving system crashes
- Large databases are the core of many companies (banks, industries, services) and must have the following three properties:
  - Resilience: The information they store is critical to the companies. If the information becomes corrupt or crashes, it can be fatal to the company.
  - High performance: It must be possible to sustain a large rate of concurrent updates while maintaining coherence
  - Scalability: The resilience and performance must be scalable to extremely large quantities of information
- How can we achieve these properties?
  - We will explain the basic principles of how to do it in this lecture

3

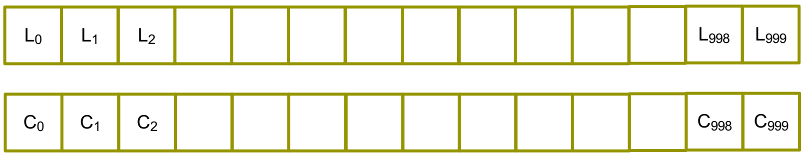
Motivating example (1)



- Consider a database represented as a large array of cells
- Many clients wish to update the database concurrently
- A naïve implementation uses one lock to protect the whole array
  - This makes it slow: only one client can modify the database at a time

4

Motivating example (2)



- A smarter implementation would use one lock per cell
- But what happens if there are two updates that conflict?
  - Each cell stores the amount in one bank account
  - First update T_1 modifies C_1 and C_2 to transfer money from C_1 to C_2
  - Second update T_2 modifies C_2 and C_3 to transfer money from C_3 to C_2
- The updates might overlap:
  - T_1 reads C_1, T_2 reads C_3 & writes C_2, T_1 writes C_2
  - This means the value in C_2 is incorrect!

Updates must be atomic and isolated!

5

The need for transactions

- The updates can do many reads and writes: large atomic actions
  - Each update must not see the intermediate states of other updates
- The system can crash at any time: actions can commit or abort
  - The database is stored permanently on disk
  - How can we ensure that the disk is not corrupted?
  - Each update stores all its effects in one place on disk
  - When the update is complete, it writes one word to disk to switch the content from the old to the new value ("one word write" must be atomic)
  - If there is a crash during the operation, the old value is restored
- The system must be fast: actions can execute in parallel
  - We allow multiple updates to execute in parallel
  - We keep temporary copies of the updates in fast memory

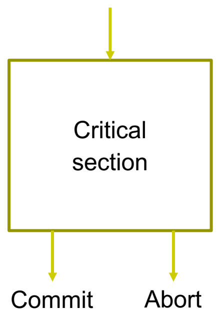
6

ACID properties

- The properties of transactions are codified in the acronym "ACID"
- A is Atomic: the transaction updates the database as if it were a single operation, either all changes are performed or none are performed (commit or abort)
  - For example, a disk crash should cause an abort with no data corruption
- C is Consistency: the update respects the invariants of the database
  - For example, total money in a bank should not change upon money transfers
- I is Isolation: two concurrent updates do not interfere, it is as if the two transactions execute in a sequential order (also called Serializability)
- D is Durability: once a transaction is committed, it survives system crashes (also called Persistence)

7

Lightweight transactions (ACI)



- An important variation is to use transactions as a programming abstraction, when durability (persistence) is not needed
  - A transaction is simply an abortable atomic action. When it aborts, all state is restored to the initial value.
  - It can abort due to internal causes (some program invariant is not satisfied) or external causes (failure outside of the program, like a file disappearing)
- Lightweight transactions are general program abstractions that can be used to replace locks when there is a chance of abort
  - Compared to exceptions, the advantage is that all the variables are restored to their initial values
- This is often called STM (Software Transactional Memory)

8

Concurrency control definition

9

Concurrency control definition

- The techniques used to build concurrent systems with transactional properties are called concurrency control
  - Concurrency control is a big and complex area with many techniques used to give many different desired properties
  - We give an introduction to the basic concepts of concurrency control
- We give one practical algorithm for implementing ACI transactions:
  - "**Optimistic** concurrency control with **strict two-phase locking** and **deadlock avoidance**"
  - We will explain all the highlighted parts in this phrase
- The textbook gives a full implementation in two pages of Oz code
  - It is the most complicated algorithm in the book!

10

Concurrency control basic concepts

- Axes of variation:
  - Optimism versus pessimism: how to give locks depending on the cost of failure
  - Lock management: how to give locks to guarantee serializability
  - Deadlock management: how to give locks to avoid circular dependencies
- Two kinds of properties:
  - Safety: never do anything wrong (e.g., system invariant)
  - Liveness: make progress (e.g., no starvation)
- Primitive building blocks:
  - Locks: control access to entities (important for safety)
  - Timestamps: give priorities to operations (important for liveness)

11

Safety and liveness

12

Safety and liveness properties

- Correctness of a system always expressed in terms of
  - safety and liveness
- Intuitively:
  - Safety properties
    - Properties that state that "something bad never happens"
  - Liveness properties
    - Properties that state that "something good eventually happens"

13

Safety & liveness are sufficient

- A property P(E) is a function of an execution E that returns true or false
  - Execution E = $(ST_0,\sigma_0) \rightarrow (ST_1,\sigma_1) \rightarrow \ldots \rightarrow (ST_i,\sigma_i) \rightarrow \ldots$
  - An execution potentially has an infinite number of execution states (ST_i,σ_i)
- It can be proved mathematically that
  - "Any [property] can be expressed as the conjunction of a safety property and a liveness property"
    (Alpern & Schneider, Inf. Proc. Letters 1985)

14

Prefixes and extensions

- To define safety and liveness precisely, we introduce the concepts of prefix and extension

$(ST_0,\sigma_0) \to (ST_1,\sigma_1) \to \ldots \to (ST_{k-1},\sigma_{k-1}) \to (ST_k,\sigma_k) \to \ldots$ Execution

$(ST_0,\sigma_0) \to (ST_1,\sigma_1) \to \ldots \to (ST_{k-1},\sigma_{k-1})$ Prefix

$(ST_0,\sigma_0) \to (ST_1,\sigma_1) \to \ldots \to (ST_{k-1},\sigma_{k-1}) \to (ST'_k,\sigma'_k) \to (ST'_{k+1},\sigma'_{k+1}) \to \ldots$ Extension

- A prefix of execution E is the first *k* (for some *k*>0) execution states
  - I.e., cut off the tail of E, finite beginning of E
- An extension of a prefix is any execution that starts with the prefix
  - The extension continues P

Safety formally defined

- Informally, property P(E) is a safety property if
  - Every execution E violating P "goes bad", i.e., it has a bad event such that every execution starting like E and behaving like E up to the bad event (including), will violate P regardless of what it does afterwards
  - When an execution "goes bad", it "stays bad"!
- Formally, a property P is a safety property if
  - Given an execution E such that P(E)=false
  - Then there exists a prefix of E such that every extension of that prefix gives an execution F with P(F)=false
    - The system "breaks" in the prefix and can't be fixed!

Safety example

- Message communication between agents on an unreliable network
  - Safety property:
    - A message sent is delivered at most once
  - Take an execution where a message is delivered more than once
    - Cut off the tail after the second delivery
    - Any extension will give an execution which also violates the property

17

Liveness formally defined

- A property P(E) is a liveness property if
  - Given any execution E such that P(E)=true,
  - Then for every prefix F of E, there exists an extension of F such that P(F)=true
- “As long as there is life there is hope”

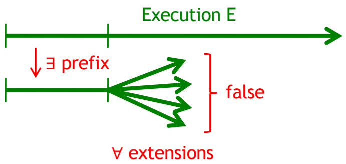
18

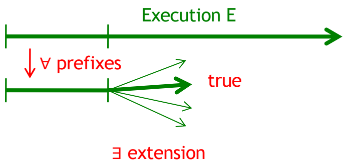
Liveness example

- Message communication between agents on an unreliable network
  - Liveness property P:
    - A message sent is delivered at least once
  - Take the prefix of any execution
    - If prefix contains delivery, any extension satisfies P
    - If prefix doesn't contain the delivery, extend it so that it contains a delivery, then the prefix + extended part will satisfy P

19

Formal definitions visually





- Safety is false for an execution E then there exists a prefix such that all extensions are false (system is broken)
- If safety is false, then this can always be shown in finite time
- Liveness is true for an execution E then for all prefixes there exists an extension that is true (system can progress)
- If liveness is false, then this can only be shown in infinite time

20

More on safety

- Safety can only be
  - satisfied in infinite time (you're never safe)
  - violated in finite time (when the bad happens)
- Often involves the word "never", "at most", "cannot",…
- Sometimes called "partial correctness"

21

More on liveness

- Liveness can only be
  - satisfied in finite time (when the good happens)
  - violated in infinite time (there's always hope)
- Often involves the words "eventually", "must", "at least"
  - Eventually means at some (often unknown) point in "future"
- Liveness is often just "termination"

22

Questions on safety and liveness

- Why not define safety to be a predicate true in every execution state?
  - Assume we have an execution $E = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_i \rightarrow \ldots$, where $e_i = (ST_i, \sigma_i)$
  - We defined safety as a property of the whole execution: P(E)
  - Why not define safety for execution states, like this: $\forall i.P(e_i)$?
  - (you tell me!)

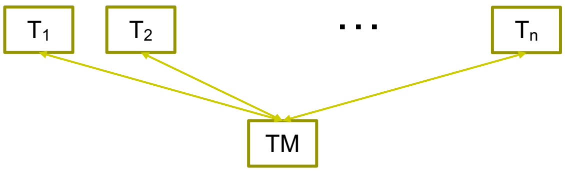
- Is every property really either liveness or safety?
  - For example, consider: “every message should be delivered exactly 1 time”
  - Is this safety or liveness?
  - (you tell me!)

23

Concurrency control concepts

24

Transaction system architecture



- Each transaction T_i runs in one thread and communicates with TM
- Concurrency control is implemented in the transaction manager TM
  - When a transaction T_i needs a lock, it asks the TM
  - The TM can give the lock, delay giving it, or refuse to give it (say no)
  - When a transaction T_i no longer needs a lock, it tells the TM

Concurrency control basic concepts

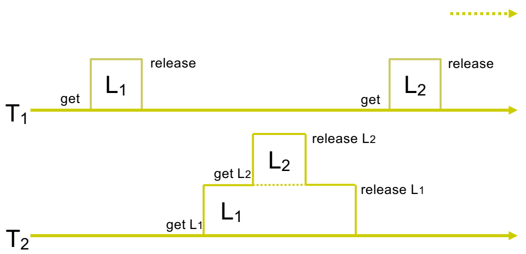
- Axes of variation:
  1. Optimism versus pessimism: how to give locks depending on the cost of failure
  2. Lock management: how to give locks to guarantee serializability
  3. Deadlock management: how to give locks to avoid circular dependencies
- Properties:
  - Safety: never do anything wrong (e.g., system invariant)
  - Liveness: make progress (e.g., no starvation)
- Primitive building blocks:
  - Locks: control access to entities (important for safety)
  - Timestamps: give priorities to operations (important for liveness)

1. Optimism versus pessimism

- When a transaction starts, it asks for locks on the items it updates
- The transaction manager has a choice to give the lock or not
  - If aborts are extremely undesirable, it will be pessimistic
  - If more transactions mean more profits, it will be optimistic
- Airline booking is an example of optimistic scheduling
  - A passenger booking a seat is a transaction
  - Airlines overbook flights, i.e., sell more tickets than seats on the plan, to increase the average number of filled seats on a flight
- Railway track allocation is an example of pessimistic scheduling
  - A train reserving a track segment is a transaction
  - Signaling mechanisms wait until it is absolutely sure that there is no other train on the same segment

27

2. Lock management: naïve solution

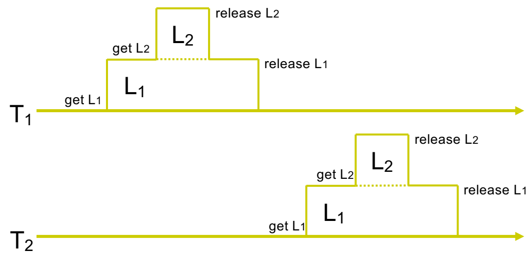


T_1 is not before T_2 and T_2 is not before T_1, so T_1 & T_2 not serializable

- Transaction T_1 transfers money between two of my accounts C_1 to C_2: it locks L_1 and decreases C_1 and then locks L_2 and increases C_2.
- Transaction T_2 calculates the total money I have in both my accounts C_1 and C_2: it locks L_1 and L_2
- Surprise! I am missing some money. Where is it?

28

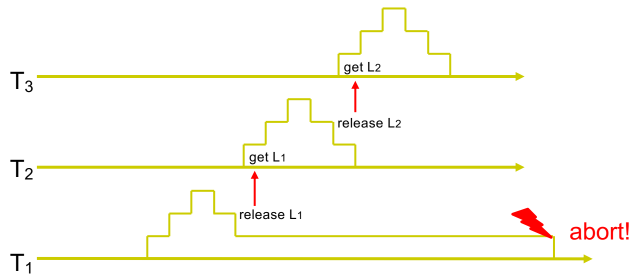
Lock management: two-phase locking



- How can we make lock management serializable?
  - When we execute transactions concurrently, they must see the data as if they were executed sequentially in some order
- We can guarantee this by using two-phase locking:
  - A transaction has two phases, a growing phase in which it only asks for locks, and a shrinking phase in which it only releases locks

29

Problem of cascading abort

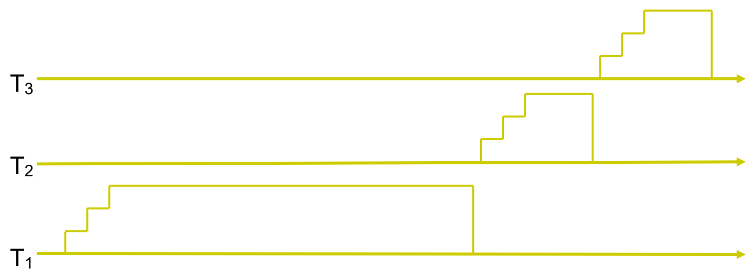


- Assume three dependent transactions T_1, T_2, T_3:
  - T_2 takes L_1 from T_1 and then completes
  - T_3 takes L_2 from T_2 and then completes
  - T_1 keeps going and eventually aborts!
- This makes the system complicated

The value of C_1 is restored.
So T_2 and T_3 must abort as well!
This is complicated!

30

Lock management: strict two-phase locking



- Cascading abort can happen when transactions have dependencies
- Two-phase locking is still correct in this case, but hard to implement!
- To avoid cascading abort, we do strict two-phase locking
  - All locks are released at once in the shrinking phase

31

A simple transaction system

32

A naïve transaction manager

- Let us design a simple transaction system that does optimistic concurrency control with strict two-phase locking
- We start with a naïve algorithm for the transaction manager:
  - When a transaction requests a lock of an unlocked cell, it gets it
  - If the cell is already locked, the transaction waits until it is unlocked
  - When a transaction commits or aborts, it releases all its locks
- The naïve algorithm does strict two-phase locking
  - It is optimistic because it assumes that getting the lock will not lead to problems later on
  - But it has a big problem: the naïve algorithm suffers from deadlock!

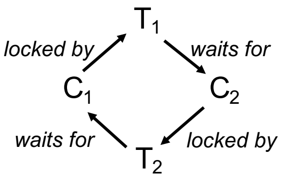
33

Deadlock example

- The naïve algorithm suffers from deadlocks
- Consider transactions T_1 and T_2 where each one uses both C_1 and C_2
  - T_1 uses C_1 first and then C_2
  - T_2 uses C_2 first and then C_1
- It can happen that T_1 has C_1's lock and T_2 has C_2's lock
  - How can this happen? Exercise!
- Then T_1 waits until T_2 releases C_2 and T_2 waits until T_1 releases C_1
  - They will both wait forever!
- How can we solve this problem?
  - Let us first understand exactly what is going on
  - Then we will fix the naïve algorithm

34

Deadlock definition



Cycle with two transactions
(it is possible to have cycles with three, four, or more transactions)

- Deadlock can happen in any system where active entities (like transactions) need resources (like cells)
  - A deadlock is a cycle in the wait-for graph
- Wait-for graph
  - Transaction nodes and cell nodes
  - Arrows from cells to transactions that lock it
  - Arrows from transactions to cells that they wait for

35

Deadlock discussion

- Deadlocks can happen in the real world
  - For example, cars at an intersection, where the active entity is a car and the resource is a square on the intersection
- In big transaction systems, deadlocks can exist for a long time without anyone becoming aware of it
  - Assume for a big bank that there are hundreds of transactions per second
  - If four transactions are in deadlock, it does not stop the others
  - But the deadlock means that part of the system is not working
- How do we solve this problem?
  - Like for diseases, there are two possibilities, namely prevention or cure, which are called deadlock avoidance and deadlock detection

36

A correct transaction manager

- We will modify the naïve algorithm to do deadlock avoidance
- We will use transaction priorities
  - Earlier transactions willl have higher priority than later transactions
  - When a transaction tries to get a lock, and the lock is already taken:
    - We compare priorities of the two transactions
    - If a lower priority has the lock, it is restarted and the lock is given to the higher
    - If a higher priority has the lock, the lower priority waits
- We can prove by induction that no deadlocks will occur
  - The first transaction has highest priority and will always continue
  - When it terminates, the next transaction has highest priority

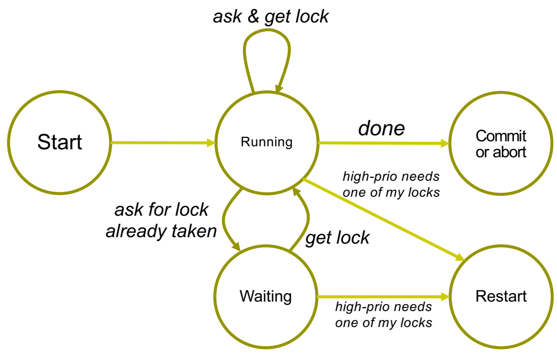
37

A correct algorithm

- A new transaction is given lower priority than all active transactions
- When a transaction tries to get a lock:
  - If the cell is unlocked, give the lock immediately
  - If the cell is locked by a transaction with higher priority, then just wait
  - If the cell is locked by a transaction with lower priority:
  - Restart the low priority transaction (forcibly abort and start again with same priority)
    - The high priority transaction then gets the lock and continues
- When a transaction commits, it releases its locks and dequeues one waiting transaction per lock
- When a transaction aborts, it restores all states and releases its locks, and dequeues one waiting transaction per lock

38

State diagram



- This state diagram shows one incarnation of the algorithm
  - Transactions may go through several incarnations until they commit or abort
  - A restart is the start state of the next incarnation

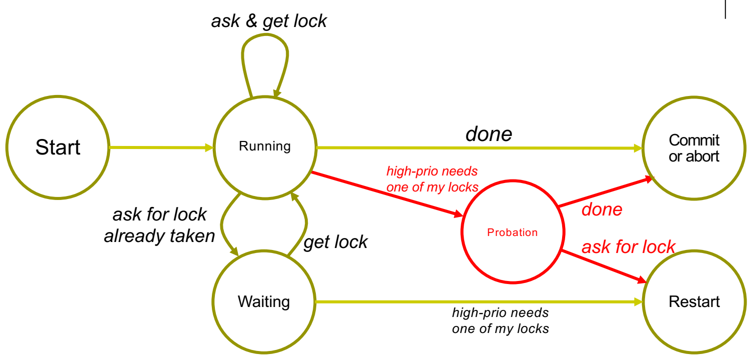
39

An improved algorithm

- The correct algorithm has a nasty implementation problem
  - It terminates running transactions at an arbitrary point during their execution
  - If done poorly can lead to inconsistencies in the run-time data structures
- It is better to terminate the transaction at a well-defined point
  - For example, when the transaction asks the manager for a lock
- We refine the algorithm to restart transactions at well-defined points
  - Instead of restarting a low-priority transaction immediately, we mark it
  - Later, when it tries to get a lock, if it is marked then it restarts

40

Improved algorithm



- A transaction in “probation” state is not allowed to get locks
  - If it tries to get a lock, it restarts

Implementation of the transaction system

Transaction abstraction (Section 8.5)

- We define an abstraction for doing ACI transactions on cells
- {NewTrans Trans NewCellT} : creates a new transaction manager and returns two operation, namely Trans for creating transactions and NewCellT for creating new cells
- {NewCellT X C} : creates a new cell C with initial value X
- {Trans **fun** {$ T} <expr> **end** B} :
  - Execute <expr> as a transaction, when it is done then B is bound to commit or abort
  - Four cell operations can be performed inside <expr>:
    - T.access, T.assign, T.exchange do the standard three cell operations
    - T.abort is a zero-argument procedure that aborts immediately
  - There are only two ways a transaction can abort: raise an exception or call T.abort

Live example of transaction manager execution

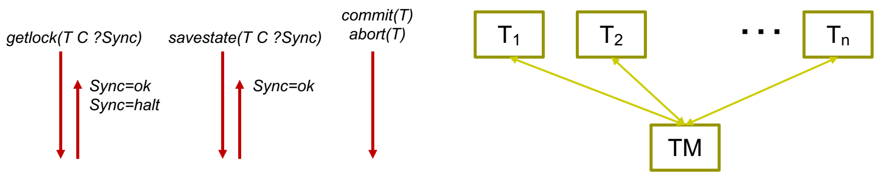
43

Implementation

- The transaction system is implemented with active objects
  - Each transaction runs in one thread and has one active object
  - The transaction manager is implemented as one active object
- Each transaction sends messages to the manager
  - getlock(T C ?Sync) : asks for lock on C, returns Sync=ok or Sync=halt
  - savestate(T C ?Sync) : saves state of C, returns Sync=ok
  - commit(T) : unlocks all T's cells and keeps their state
  - abort(T) : unlocks all T's cells and restores their state
- The transaction manager has three roles:
  - Managing the cell locks: giving or refusing them (refusal causes restart)
  - Managing the cell states: saving and restoring them
  - Managing the transactions: handling commit, abort, restart

44

Transaction system message protocol



- Each transaction sends messages to the transaction manager
  - Possible messages: ask for lock, save cell state, commit, abort
- The transaction manager keeps track of the cell locks, the cell states, and the transaction states

45

Transaction record

- Each transaction has a record containing all its local data:

  T=trans(stamp:TS save:D body:P state:{NewCell running} result:R)

- stamp : timestamp (integer priority, lower value is higher priority)
- save : dictionary containing saved cell states where each key is a cell name and the value is the record save(cell:C state:S)
- body : one-argument function transaction body
- state : element of the set {running, waiting_on(C), probation}

46

Cell record

- Each cell has a record containing all its local data:

  C=cell(name:N owner:{NewCell **unit**} queue:Q state:{NewCell X})

- name : unique constant giving the cell's name
- owner : transaction that is currently locking the cell
- queue : priority queue of transactions waiting for cell's lock
- state : cell's current state

Priority queue

- A priority queue is a queue whose entries are always ordered according to their priorities
  - Q={NewPrioQueue} : creates an empty priority queue Q
  - {Q.enqueue X P} : insert X with integer priority P
  - X={Q.dequeue} : remove and return the entry with the smallest value of P
  - X={Q.delete P} : remove and return an entry with priority P (it must exist)
  - B={Q.isEmpty} : return true or false depending on whether Q is empty

- Each locked cell has a priority queue of waiting transactions
  - The highest priority waiting transaction will get the lock when it is released

Conclusions

49

Conclusions

- Transactions are large atomic actions that can commit or abort
  - This makes them useful for building fault-tolerant systems
  - Transactions are defined by the ACID properties
- Transactions are widely used to manage large databases
  - Most companies use transactional databases to manage critical data
- Transaction systems have strong implementation constraints :
  - Safety and liveness properties
  - Resilience, performance, scalability (which safety and which liveness?)
  - Concurrency control = techniques for building transaction systems
- We show a transaction system that does optimistic concurrency control with strict two-phase locking and deadlock avoidance
  - The most complicated algorithm in the textbook!

50

```
% LINFO1131
% Advanced Programming Language Concepts

% Lecture 11 (Dec. 20, 2023)

% Transaction manager

%%%%%%%%%%%%%%%%%%%

%%%%%% Active objects
declare
fun {NewActive Class Init}
   Obj={New Class Init}
   P
in
   thread S in
      {NewPort S P}
      {ForAll S proc {$ M} {Obj M} end}
   end
   proc {$ M} {Send P M} end
end

%%%%%%%%%%%%%%%%%%%

%%%%%% Priority queue
declare
fun {NewPrioQueue}
   Q={NewCell nil}
   proc {Enqueue X Prio}
      fun {InsertLoop L}
         case L of pair(Y P)|L2 then
            if Prio<P then pair(X Prio)|L
            else pair(Y P)|{InsertLoop L2} end
         [] nil then [pair(X Prio)] end
      end
   in Q:={InsertLoop @Q} end

   fun {Dequeue}
      pair(Y _)|L2=@Q
   in
      Q:=L2 Y
   end

   fun {Delete Prio}
      fun {DeleteLoop L}
         case L of pair(Y P)|L2 then
            if P==Prio then X=Y L2
            else pair(Y P)|{DeleteLoop L2} end
         [] nil then nil end
      end X
   in Q:={DeleteLoop @Q} X end

   fun {IsEmpty} @Q==nil end
in
```

```
   queue(enqueue:Enqueue dequeue:Dequeue
         delete:Delete isEmpty:IsEmpty)
end

%%%%%%%%%%%%%%%%%%%

%%%%%% Transaction manager
declare
class TMClass
   attr timestamp tm
   meth init(TM) timestamp:=0 tm:=TM end

   meth Unlockall(T RestoreFlag)
      for save(cell:C state:S) in {Dictionary.items T.save} do
         (C.owner):=unit
         if RestoreFlag then (C.state):=S end
         if {Not {C.queue.isEmpty}} then
         Sync2#T2={C.queue.dequeue} in
            (T2.state):=running
            (C.owner):=T2 Sync2=ok
         end
      end
   end

  meth Trans(P ?R TS)
     Halt={NewName}
     T=trans(stamp:TS save:{NewDictionary} body:P
             state:{NewCell running} result:R)
     proc {ExcT C X Y} S1 S2 in
        {@tm getlock(T C S1)}
        if S1==halt then raise Halt end end
        {@tm savestate(T C S2)} {Wait S2}
        {Exchange C.state X Y}
     end
     proc {AccT C ?X} {ExcT C X X} end
     proc {AssT C X} {ExcT C _ X} end
     proc {AboT} {@tm abort(T)} R=abort raise Halt end end
  in
     thread try Res={T.body t(access:AccT assign:AssT
                              exchange:ExcT abort:AboT)}
            in {@tm commit(T)} R=commit(Res)
            catch E then
               if E\=Halt then {@tm abort(T)} R=abort(E) end
     end end
  end

  meth getlock(T C ?Sync)
     if @(T.state)==probation then
        {self Unlockall(T true)}
        {self Trans(T.body T.result T.stamp)} Sync=halt
     elseif @(C.owner)==unit then
        (C.owner):=T Sync=ok
     elseif T.stamp==@(C.owner).stamp then
        Sync=ok
```

```
    else /* T.stamp\=@(C.owner).stamp */ T2=@(C.owner) in
       {C.queue.enqueue Sync#T T.stamp}
       (T.state):=waiting_on(C)
       if T.stamp<T2.stamp then
          case @(T2.state) of waiting_on(C2) then
          Sync2#_={C2.queue.delete T2.stamp} in
             {self Unlockall(T2 true)}
             {self Trans(T2.body T2.result T2.stamp)}
             Sync2=halt
          [] running then
             (T2.state):=probation
          [] probation then skip end
       end
    end
  end

   meth newtrans(P ?R)
      timestamp:=@timestamp+1 {self Trans(P R @timestamp)}
   end
   meth savestate(T C ?Sync)
      if {Not {Dictionary.member T.save C.name}} then
         (T.save).(C.name):=save(cell:C state:@(C.state))
      end Sync=ok
   end
   meth commit(T) {self Unlockall(T false)} end
   meth abort(T) {self Unlockall(T true)} end
end

proc {NewTrans ?Trans ?NewCellT}
TM={NewActive TMClass init(TM)} in
   fun {Trans P ?B} R in
      {TM newtrans(P R)}
      case R of abort then B=abort unit
      [] abort(Exc) then B=abort raise Exc end
      [] commit(Res) then B=commit Res end
   end
   fun {NewCellT X}
      cell(name:{NewName} owner:{NewCell unit}
           queue:{NewPrioQueue} state:{NewCell X})
   end
end

%%%%%%%%%%%%%%%%%%%
```

```
% LINFO1131
% Advanced Programming Language Concepts

% Lecture 11 (Dec. 20, 2023)

% Examples of transactions

%%%%%%%%%%%%%%%%%%%

% Create transaction manager
declare Trans NewCellT in
{NewTrans Trans NewCellT}

% Create a small database
declare
D={MakeTuple db 1000}
for I in 1..1000 do D.I={NewCellT I} end

% Define transaction that mixes up numbers in database
% Sum of database entries is invariant
declare
fun {Rand} {OS.rand} mod 1000 + 1 end
proc {Mix} {Trans
                proc {$ T _}
                   I={Rand} J={Rand} K={Rand}
                   A={T.access D.I}
                   B={T.access D.J} C={T.access D.K}
                in
                   {T.assign D.I A+B-C}
                   {T.assign D.J A-B+C}
                   if I==J orelse I==K orelse J==K then
                          {T.abort} end
                   {T.assign D.K ~A+B+C}
                end _ _}
end

% Define transaction that sums all entries in database
declare
S={NewCellT 0}
fun {Sum}
   {Trans
        fun {$ T} {T.assign S 0}
           for I in 1..1000 do
                  {T.assign S {T.access S}+{T.access D.I}}
           end
           {T.access S}
        end _}
end

{Browse {Sum}}  % Displays 500500 (sum of all entries)

% Mix up the database entries with 1000 concurrent mix transactions
for I in 1..1000 do thread {Mix} end end
```

```
{Browse {Sum}}  % Still displays 500500 (sum is invariant)

% Display first 10 entries of the database
{Trans  proc {$ T _}
           for I in 1..10 do
             {Browse {T.access D.I}}
           end
        end _ _}

%%%%%%%%%%%%%%%%%%%
```