

10. Concurrent and functional programming

PPj-94c

Overview

1. Pure **functional programs do not have side-effects**: operands of an operation and arguments of a call can be **evaluated in any order**, in particular **concurrently**
2. **Recursive task decomposition** can be parallelized according to the paradigm **bag of subtasks**
3. **Lazy evaluation** of lists leads to **programs that transform streams**, can be **parallelized** according to the **pipelining** paradigm
4. **Dataflow languages** and dataflow machines support **stream programming**
5. **Concurrency notions in functional languages**:
Message passing in Erlang
Actors in Scala

© 2014 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 94c

Objectives:

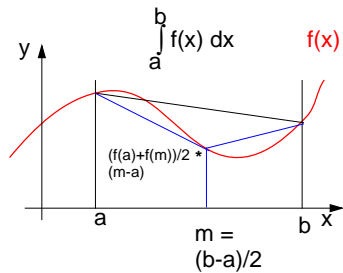
Understand close relation between FP and concurrency

In the lecture:

- Explain
- the items.

Recursive adaptive quadrature computation

PPj-94d



```
fun quad (f, l, r, area, eps) =  
  let m = (r-l)/2 and  
      fl = f(l) and  
      fm = f(m) and  
      fr = f(r) and  
      larea = (fl+fm)*(m-l)/2 and  
      rarea = (fm+fr)*(r-m)/2 and
```

```
  in  
    if abs(larea+rarea-area)>eps  
    then  
      let
```

```
        lar = quad(f,l,m,larea,eps) and
```

```
        rar = quad(f,m,r,rarea,eps)
```

```
      in (lar+rar)  
      end  
    else area  
  end
```

initial call:

```
quad (f,a,b,(f(a)+f(b))/2*(b-a),0.001)
```

Compute an **approximation of the integral** over $f(x)$ between a and b .

Recursively **refine the interval** into two subintervals until the sum of the **areas of the two trapezoids** differs less than eps from the **area of the big trapezoid**.

See [G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 17-19]

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 94d

Objectives:

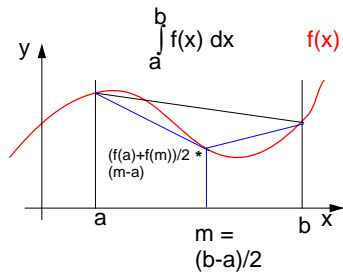
Understand the recursive quadrature computation

In the lecture:

Explain

- the task,
- the approximation idea,
- the functional notation

Recursive adaptive quadrature computation



Compute an **approximation of the integral** over $f(x)$ between a and b .

Recursively **refine the interval** into two subintervals until the sum of the **areas of the two trapezoids** differs less than eps from the **area of the big trapezoid**.

Fork two concurrent processes.

See [G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 17-19]

```

fun quad (f, l, r, area, eps) =
let
  m = (r-l)/2 and
  fl = f(l) and
  fm = f(m) and
  fr = f(r) and
  larea = (fl+fm)*(m-l)/2 and
  rarea = (fm+fr)*(r-m)/2 and
in
  if abs(larea+rarea-area)>eps
  then
  let
    co
    lar = quad(f,l,m,larea,eps) and
    //
    rar = quad(f,m,r,rarea,eps)
    oc
  in (lar+rar)
  end
  else area
end
initial call:
quad (f,a,b,(f(a)+f(b))/2*(b-a),0.001)

```

Lecture Parallel Programming WS 2014/2015 / Slide 94e

Objectives:

Parallelized refinement

In the lecture:

Explain

- the dynamics of the refinement

Streams in functional programming

Linear lists are fundamental data structures in functional programming, e.g. in **SML**:

```
datatype 'a list = nil | :: of 'a * 'a list
```

Eager evaluation: all elements of a list are to be computed, before any can be accessed.

Lazy evaluation only those elements of a list are computed which are going to be accessed.

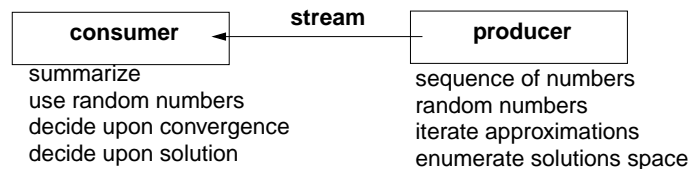
That can be achieved by replacing the (pointer to) the tail of the list by a parameterless **function which computes the tail of the sequence when needed**:

```
datatype 'a seq= Nil | Cons of 'a * (unit->'a seq)
```

Lazy lists are called **streams**.

Streams establish a useful **programming paradigm**:

Programming the **creation** of a stream can be **separated** from programming its **use**.



Functions on streams can be understood as communicating concurrent processes.

Lecture Parallel Programming WS 2014/2015 / Slide 94f

Objectives:

Understand streams in functional programming

In the lecture:

Explain

- the notion of streams.
- the programming paradigm

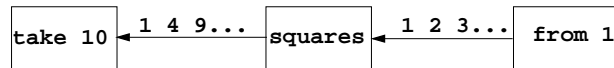
Examples for stream functions (1)

produce a stream of numbers: `int -> int seq`
`fun from k = Cons (k, fn()=> from (k+1));`

consume the first n elements into a list: `'a seq * int -> 'a list`
`fun take (xq, 0) = []`
`| take (Nil, n) = raise Empty`
`| take (Cons(x, xf), n) = x :: take (xf (), n - 1);`

transform a stream of numbers: `int seq -> int seq`
`fun squares Nil = Nil`
`| squares (Cons (x, xf)) = Cons (x * x, fn() => squares (xf()));`

`take (squares (from 1), 10);`



Lecture Parallel Programming WS 2014/2015 / Slide 94g

Objectives:

Understand simple stream functions

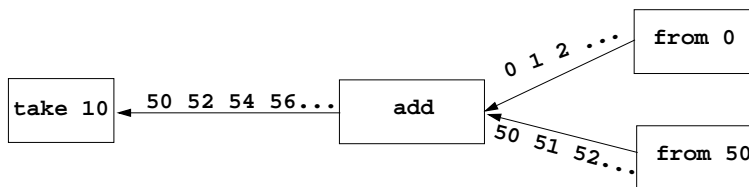
In the lecture:

Explain

- structure of stream functions,
- classify: producer, consumer, transformer,

Examples for stream functions (2)

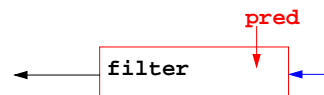
add the numbers of two streams: `int seq * int seq -> int seq`
`fun add (Cons(x, xf),Cons(y, yf)) =`
`Cons (x+y, fn() => add (xf(), yf()))`
`| add _ = Nil;`



Filter-Schema:

`('a -> bool) -> 'a seq -> 'a seq`

`fun filter pred Nil = Nil`
`| filter pred (Cons(x,xf)) =`
`if pred x then Cons (x, fn()=> filter pred (xf()))`
`else filter pred (xf());`



Lecture Parallel Programming WS 2014/2015 / Slide 94h

Objectives:

Combine streams

In the lecture:

Explain the examples

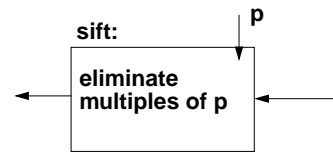
Recursive stream composition

PPJ-94i

```
fun sift p =
  filter (fn n => n mod p <> 0);
```

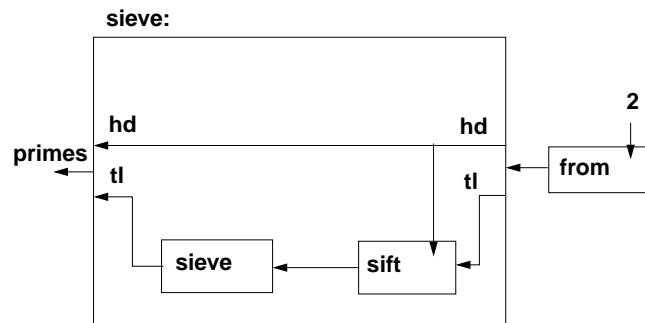
```
fun sieve (Cons(p,nf)) =
  Cons (p, fn() => sieve (sift p (nf())));
```

```
val primes = sieve (from 2);
take (primes, 25);
```



Compute prime numbers:

Sieve of Eratosthenes



All recursively constructed sift-sieve-pairs can execute concurrently!

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 94i

Objectives:

Understand recursive composition of streams

In the lecture:

The recursion is explained

Sieve of Eratosthenes in CSP

PPJ-94j

A pipeline of filters:

L processes are created, each sends a **stream of numbers** to its successor.

The **first number p** received is a prime. It is **used to filter** the following numbers.

Finally, each process holds a prime in p.

```
process Sieve[1]
  for [1 = 3 to n by 2]
    Sieve[2] ! i # pass odd numbers to Sieve[2]

process Sieve[i = 2 to L]
  int p, next
  Sieve[i-1] ? p          # p is a prime
  do Sieve[i-1] ? next -># receive next candidate
    if (next mod p) != 0 ->
      Sieve[i+1] ! next # pass it on
  fi
od
```

[G. Andrews: Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000, pp. 326-328]

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 94j

Objectives:

Pipeline of processes

In the lecture:

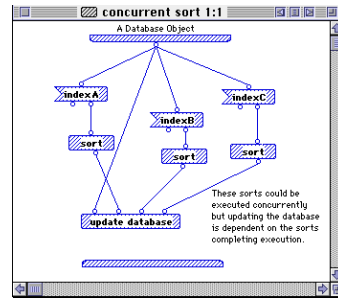
The communication is explained and compared to the stream functions.

Dataflow languages

Textual languages:

Lucid: stream computations by equations, no side effects; 1976, Wadge, Ashcroft

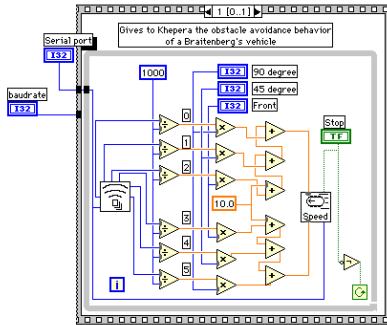
SISAL: (Streams and Iteration in a Single Assignment Language), no side-effects, fine-grained parallelization by compiler, 1983



Visual languages:

Prograph (Acadia University 1983):
dataflow and object-oriented

LabVIEW (National Instruments, 1986):
Nodes represent stream processing functions connected by wires, concurrent execution triggered by available input. Strong support of interfaces to instrumentation hardware.



Lecture Parallel Programming WS 2014/2015 / Slide 94k

Objectives:

Pointers to dataflow languages

In the lecture:

General information on these languages is given.

Language Erlang

Erlang developed 1986 by Joe Armstrong, et.al at **Ericsson**

- multi-paradigm: **functional** and **concurrent**
- initial application area: **telecommunication**
requirements: distributed, fault-tolerant, soft-real-time, non-stopping software
- **processes** communicate via **asynchronous message** passing
- **single-assignment** variables, **no shared memory** between processes

Explanations and examples taken from

[J. Armstrong, R. Virding, C. Wikström, M. Williams: Concurrent Programming in ERLANG, Second Edition, Ericsson Telecommunications Systems Laboratories, Prentice Hall, 1996]

<http://www.erlang.org>

Lecture Parallel Programming WS 2014/2015 / Slide 94l

Objectives:

Characteristics of Erlang

In the lecture:

Explain background

Basic communication constructs

process creation:

```
Pid = spawn(Module, FunctionName, ArgumentList)
```

asynchronous message send:

```
Pid ! Message
```

The operands are expressions which yield a process id and a message.

selective receive:

```
receive
  Pattern1 [when Guard1] ->
    Actions1 ;
  Pattern2 [when Guard2] ->
    Actions2 ;
  ...
end
```

Searches the process' **mailbox** for a **message that matches a pattern**, and receives it.
Can not block on an unexpected message!

Initial example

A module that creates counter processes:

```
-module(counter).
-export([start/0,loop/1]).

start() ->
  spawn(counter, loop, [0]).

loop(Val) ->
  receive
    increment ->
      loop(Val + 1)
  end.

clients send increment messages to it
```

Lecture Parallel Programming WS 2014/2015 / Slide 94m

Objectives:

Understand send and selective receive constructs

In the lecture:

Explain the constructs

Complete example: Counter

Interface functions are called by client processes.

They send 3 kinds of messages.

`self()` gives the client's pid, to reply to it.

The counter process identifies itself in the reply.

The receive is **iterated** (tail-recursion).

Unexpected messages are **removed**

```
-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% First the interface functions.
start() -> spawn(counter, loop, [0]).

increment(Counter) -> Counter ! increment.

value(Counter) ->
  Counter ! {self(),value},
  receive {Counter,value} -> Value
end.

stop(Counter) -> Counter ! stop.

%% The counter loop.
loop(Val) ->
  receive increment ->    loop(Val + 1);
           {From,value} -> From ! {self(),Val},
           loop(Val);
           stop ->        true;
           Other ->       loop(Val)
end.
```

Lecture Parallel Programming WS 2014/2015 / Slide 94n

Objectives:

Understand the constructs in context

In the lecture:

Explain the constructs

Example: Allocation server (interface)

A server maintains two lists of **free and allocated resources**. Clients call a function **allocate** to request a resource and a function **free** to return that resource.

The two lists of **free and allocated resources** are initialized.

register associates the pid to a name.

The calls of **allocate** and **free** are transformed into **different kinds of messages**. Thus, implementation details are not disclosed to clients.

```
-module(allocator).
-export([start/1,server/2,allocate/0,free/1]).

start(Resources) ->
    Pid = spawn(allocator, server,
                [Resources,[]]),
    register(resource_alloc, Pid).

% The interface functions.

allocate() -> request(alloc).

free(Resource) -> request({free,Resource}).

request(Request) ->
    resource_alloc ! {self(),Request},
    receive {resource_alloc,Reply} -> Reply
end.
```

Lecture Parallel Programming WS 2014/2015 / Slide 94o

Objectives:

Understand a non-trivial server implementation

In the lecture:

Explain the techniques

Example: Allocation server (implementation)

The function **server** receives the two kinds of messages and transforms them into calls of **s_allocate** and **s_free**.

s_allocate returns **yes** and the resource or **no**, and **updates** the two lists in the recursive **server** call.

s_free: **member** checks whether the returned resource **R** is in the free list, returns **ok** and updates the lists,

or it returns **error**.

The **server** call loops.

```
server(Free, Allocated) ->
    receive
        {From,alloc} ->
            s_allocate(Free, Allocated, From);
        {From,{free,R}} ->
            s_free(Free, Allocated, From, R)
    end.

s_allocate([R|Free], Allocated, From) ->
    From ! {resource_alloc,{yes,R}},
    server(Free, [[R,From]|Allocated]);
s_allocate([], Allocated, From) ->
    From ! {resource_alloc,no},
    server([], Allocated).

s_free(Free, Allocated, From, R) ->
    case member({R,From}, Allocated) of
        true -> From ! {resource_alloc,ok},
            server([R|Free],
                    delete({R,From},
                            Allocated));
        false ->From ! {resource_alloc,error},
            server(Free, Allocated)
    end.
```

Lecture Parallel Programming WS 2014/2015 / Slide 94p

Objectives:

Understand a non-trivial server implementation

In the lecture:

Explain the techniques

Scala: object-oriented and functional language

Scala: Object-oriented language (like Java, more compact notation), augmented by functional constructs (as in SML); object-oriented execution model (Java)

functional constructs:

- nested functions, higher order functions, currying, case constructs based on pattern matching
- functions on lists, streams,... provided in a big language library
- parametric polymorphism; restricted local type inference

object-oriented constructs:

- classes define all types (types are consequently oo - including basic types), subtyping, restrictable type parameters, case classes
- object-oriented mixins (traits)

general:

- static typing, parametric polymorphism and subtyping polymorphism
- very compact functional notation
- complex language, and quite complex language description
- compilable and executable together with Java classes
- since 2003, author: Martin Odersky, www.scala.org, docs.scala-lang.org

Lecture Parallel Programming WS 2014/2015 / Slide 94q

Objectives:

Overview over properties of Scala

In the lecture:

Brief explanations are given

Actors in Scala (1)

An **actor** is a lightweight process:

- **actor** { **body** } creates a process that executes **body**
- **asynchronous** message passing
- **send**: **p ! msg** puts **msg** into **p**'s mailbox
- **receive** operation searches the mailbox for the first message that matches one of the case patterns (as in **Erlang**)
- **case x** is a catch-all pattern

Example: orders and cancellations

```
val orderMgr = actor {
  while (true)
    receive {
      case Order(sender, item) =>
        val o =
          handleOrder(sender, item)
        sender ! Ack(o)
      case Cancel(sender, o) =>
        if (o.pending) {
          cancelOrder(o)
          sender ! Ack(o)
        } else sender ! NoAck
      case x => junk += x
    }
}

val customer = actor {
  orderMgr ! Order(self, myItem)
  receive {
    case Ack(o) => ...
  }
}
```

[P. Haller, M. Odersky: Actors That Unify Threads and Events; in A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 171–190, 2007. © Springer-Verlag Berlin Heidelberg 2007]

Lecture Parallel Programming WS 2014/2015 / Slide 94r

Objectives:

Understand actors in Scala

In the lecture:

Explain

- the constructs using the example,
- asynchronous message passing via unordered mailbox
- sender is explicitly transmitted for reply: common pattern

Actors in Scala (2)

Constructs used to simplify replying:

- The sender of a received message is stored in `sender`
- `reply(msg)` sends `msg` to `sender`
- `a !? msg` sends `msg` to `a`, waits for a reply, and returns it.

Example: orders and cancellations

```
val orderMngr = actor {
  while (true)
    receive {
      case Order(item) =>
        val o =
          handleOrder(sender,item)
        reply(Ack(o))
      case Cancel(o) =>
        if (o.pending) {
          cancelOrder(o)
          reply(Ack(o))
        } else reply(NoAck)
      case x => junk += x
    }
}

val customer = actor {
  orderMngr !? Order(myItem)
  match {
    case Ack(o) => ...
  }
}
```

[P. Haller, M. Odersky: Actors That Unify Threads and Events; in A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 171–190, 2007. © Springer-Verlag Berlin Heidelberg 2007]

Lecture Parallel Programming WS 2014/2015 / Slide 94s

Objectives:

Scala provides simplifying notations

In the lecture:

Explain the simplifying constructs

11. Check your knowledge (1)

Introduction

1. Explain the notions: sequential, parallel, interleaved, concurrent execution of processes.
2. How are Threads created in Java (3 steps)?

Properties of Parallel Programs

3. Explain axioms and inference rules in Hoare Logic.
4. What does the weakest precondition $wp(s, Q) = P$ mean?
5. Explain the notions: atomic action, at-most-once property.
6. How is interference between processes defined?
7. How is non-interference between processes proven?
8. Explain techniques to avoid interference between processes.

Monitors

9. Explain how the two kinds of synchronization are used in monitors.
10. Explain the semantics of condition variables and the variants thereof.
11. Which are the 3 reasons why a process may wait for a monitor?
12. How do you implement several conditions with a single condition variable?

Lecture Parallel Programming WS 2014/2015 / Slide 95

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (2)

PPJ-95a

13. Signal-and-continue requires loops to check waiting-conditions. Why?
14. Explain the properties of monitors in Java.
15. When can notify be used instead of notifyAll?
16. Where does a monitor invariant hold? Where has it to be proven?
17. Explain how monitors are systematically developed in 5 steps.
18. Formulate a monitor invariant for the readers/writers scheme?
19. Explain the development steps for the method „Rendezvous of processes“.
20. How are waiting conditions and release operations inserted when using the method of counting variables?

Barriers

21. Explain duplication of distance at the example prefix sums.
22. Explain the barrier rule; explain the flag rules.
23. Describe the tree barrier.
24. Describe the symmetric dissemination barrier.

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 95a

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (3)

PPJ-96

Data parallelism

25. Explain how list ends are found in parallel.
26. Show iteration spaces for given loops and vice versa.
27. Explain which dependence vectors may occur in sequential (parallel) loops.
28. Explain the SRP transformations.
29. How are the transformation matrices used?
30. Which transformations can be used to parallelize the inner loop if the dependence vectors are $(0,1)$ and $(1,0)$?
31. How are bounds of nested loops described formally?

Asynchronous messages

32. Explain the notion of a channel and its operations.
33. Explain typical channel structures.
34. Explain channel structures for the client/server paradigm.
35. What problem occurs if server processes receive each from several channels?
36. Explain the notion of conversation sequences.

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 96

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (4)

PPJ-97

37. Which operations does a node execute when it is part of a broadcast in a net?
38. Which operations does a node execute when it is part of a probe-and-echo?
39. How many messages are sent in a probe-and-echo scheme?

Messages in distributed systems

40. Explain the worker paradigm.
41. Describe the process interface for distributed branch-and-bound.
42. Explain the technique for termination in a ring.

Synchronous messages

43. Compare the fundamental notions of synchronous and asynchronous messages.
44. Explain the constructs for selective wait with synchronous messages.
45. Why are programs based on synchronous messages more compact and less redundant than those with asynchronous messages?
46. Describe a server for resource allocation according to the scheme for synchronous messages.

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 97

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.

Check your knowledge (5)

PPJ-98

Concurrent and functional programming

47. Explain why paradigms in functional and concurrent programming match well.
48. What are benefits of stream programming?
49. Compare implementations of the Sieve of Eratosthenes using streams or CSP.
50. Explain concurrency in Erlang, in particular selective receive.
51. Explain the characteristics of Scala, in particular its Actors.

© 2013 bei Prof. Dr. Uwe Kastens

Lecture Parallel Programming WS 2014/2015 / Slide 98

Objectives:

Understand and repeat the material

In the lecture:

- Answer some of the questions.