TDDA69 Data and Program Structure
Declarative Concurrency

Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

Lecture content

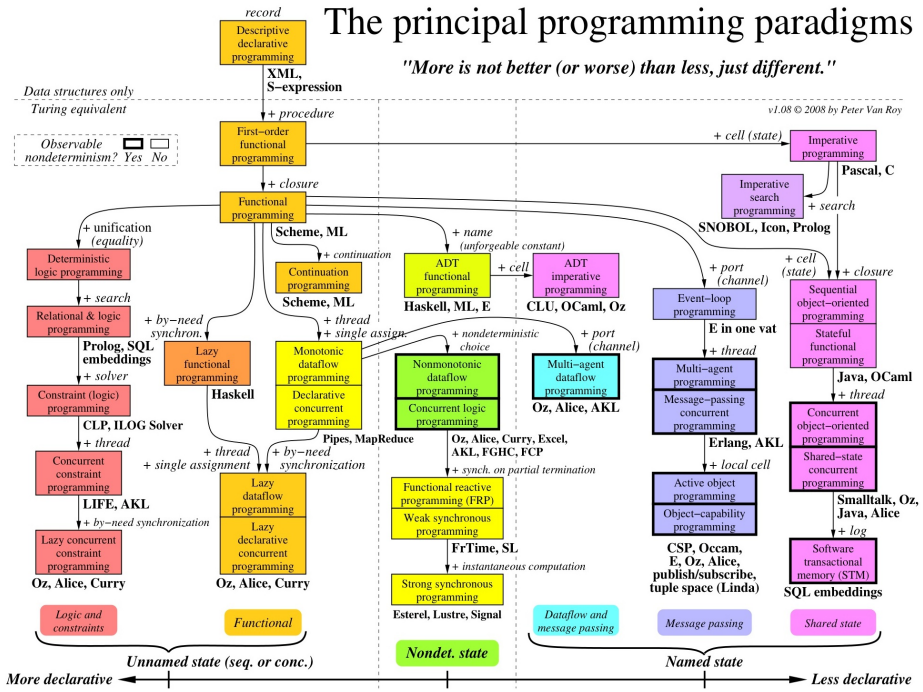
- Data-driven concurrent model
- Lazy execution
- MapReduce
- Stream

Concurrency

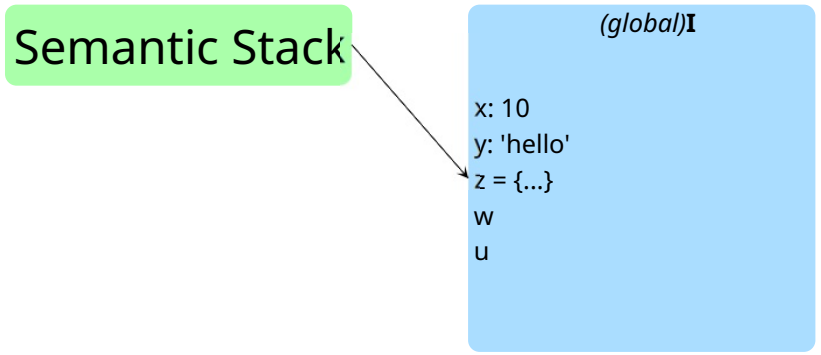
- Natural for some applications:
  - GUI: event thread, computation threads...
  - Games: graphics thread, ai thread...
  - Operating Systems: driver threads...
  - Data processing...
- Distributed applications: client-server

Declarative Concurrency

- No observable nondeterminism
  - The result is only dependent of the inputs
- Communication through shared dataflow variables

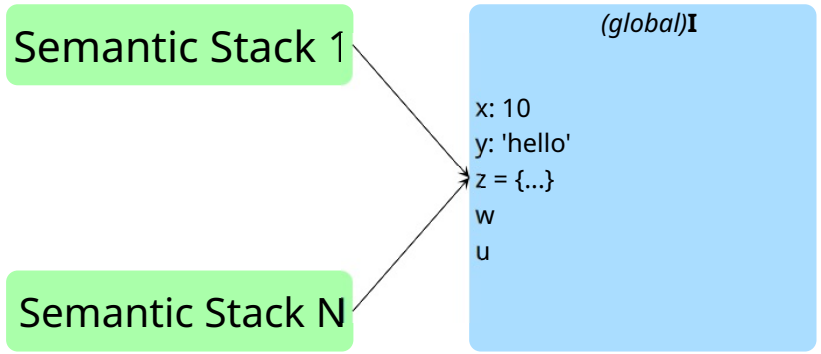


Sequential Model



Data-driven concurrent model

Concurrent Model



Extension to the syntax of KL

- STATEMENT := (... | THREAD)
- THREAD := 'thread' '(' (EXPR (, EXPR)+)) | () ')' ('->' IDENTIFIER) BLOCK
- Example:

```
define my_thread = thread(a) -> b {
  b = fib(a);
};
define v1 = my_thread(1000);
define v2 = my_thread(2000);
std.print(v1 + v2);
```


Basic concepts

- The model allows multiple statements to execute "at the same time"
- Reading and writing different variables can be done simultaneously by different threads, as well as reading the same variable
- What about writting to the same variable?

Deterministic vs Nondeterministic

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there are multiple concurrent states
- A declarative language needs to be deterministic

Example of non determinism

```
define x;
thread() {
  x = 1;
}();
thread() {
  x = 3;
}();
```

- The first thread that bind x succeed, the other one fails
- Commonly called "data race".

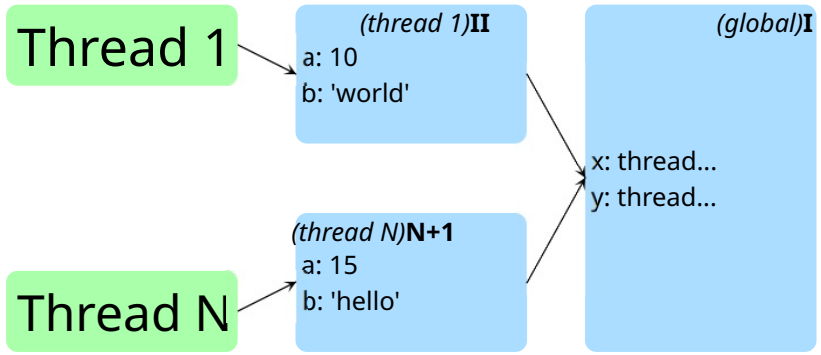
How to solve?

- Use exception?

```
define x;
thread() {
  catch(x = 1);
}();
thread() {
  catch(x = 3);
}();
```

- Not deterministic!
- Two solutions:
  - Leave it as an error to solve by the programmer
  - An alternative model is to assign a value store for each thread

Multiple value stores



Thread synchronisation

- Implicit synchronisation

  Wait for the computation to be available

```
define my_thread = thread(a) -> b {
  b = fib(a);
};
define v1 = my_thread(1000);
define v2 = my_thread(2000);
std.print(v1 + v2);
```

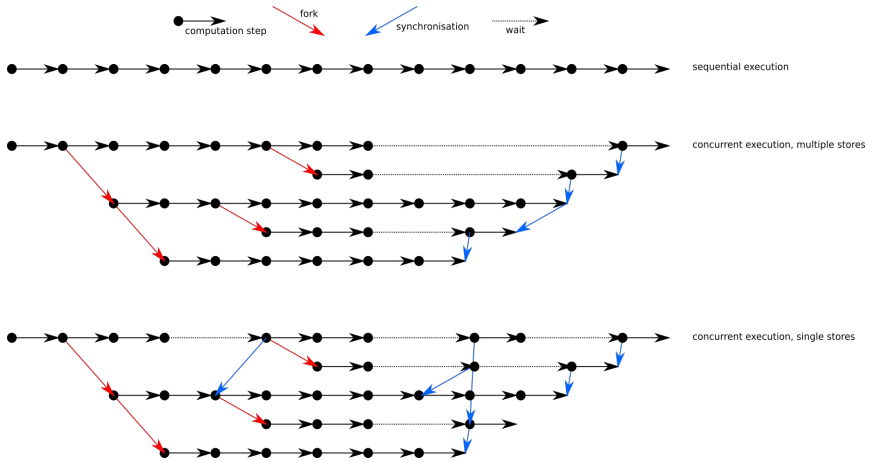
  The main thread is blocked until v1 and v2 are finished.
- v1 and v2 are called *dataflow variable*

Dead-lock

```
define thread_1 = thread() {
  ...
  ... thread_2 ...;
  ...
}();
define thread_2 = thread() {
  ...
  ... thread_1 ...;
  ...
}();
```

- But deterministic...

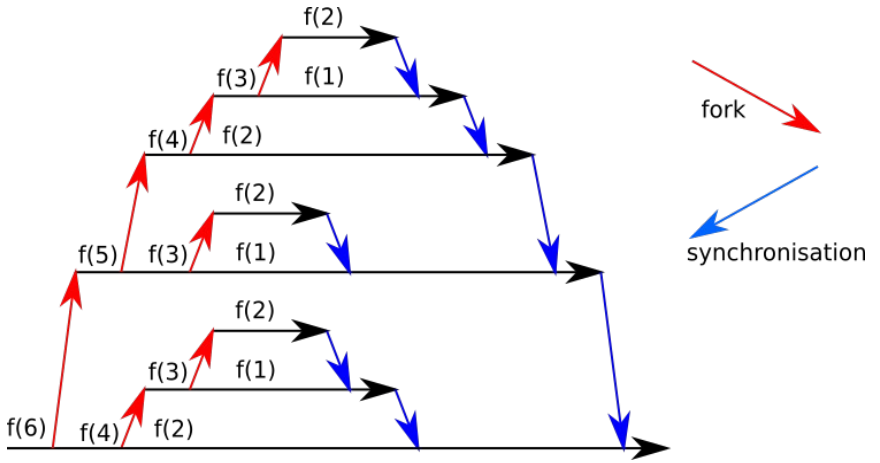
Sequential / Concurrent Execution



Since concurrency is easy...

- Declarative programs can be easily made concurrent
- There is no real drawbacks (except cost of starting a thread)
-
```
define fib = function(n) -> v {
  v = cond(n <= 2, 1,
           fib(x-1) + fib(x-2));
}
```
-
```
define fib = function(n) -> v {
  v = cond(n <= 2, 1,
          (thread() -> v { v = fib(x-1) })()
              + fib(x-2));
}
```


Execution of threaded fib(6)



Lazy execution

Lazy execution

- Up to now the execution order of each thread follows textual order, when a statement comes as the first in sequence it will execute, whether or not its results are needed later
- The execution scheme is called **eager execution**, or **supply-driven** execution
- Another execution order is that a statement is executed only if its results are needed some in the program
- This scheme is called **lazy evaluation**, or **demand-driven** evaluation

Example

- ```
  define B = f1(X);
  define C = f2(X);
  define A = B + C;
  ```
- If f1 and f2 are pure, delay evaluation of B and C until it is needed to evaluate A

Lazy Evaluation and Stor

- Without lazy

```
define fib =
function(n) -> r { ...
};
define a = fib(10);
```

- With lazy

```
define fib =
function(n) -> r { ...
};
define a = fib(10);
a
```

store not lazy

fib: function(n) -> r { ... }
a: 55

store lazy

fib: function(n) -> r { ... }
a: 55
```

Concurrency

```
define my_thread =
thread(a) -> b {
  b = fib(a);
};

define v1 =
my_thread(1000);
define v2 =
my_thread(2000);
std.print(v1 + v2);
```

```
define v1 =
fib(1000);
define v2 =
fib(2000);
std.print(v1 + v2);
```


MapReduce

Map/Reduce

- Map is a higher-order function that applies a given functor to each element of a list

```
map([1, 2, 3, 4, 5], square) -> [1, 4, 9, 16, 25]
```

- Reduce is a higher-order function that applies a reduction to a data set

```
reduce([1, 2, 3, 4, 5], add) -> 15
```


Concurrent Map

- Split the dataset
-
```
function tmap(list, functor) -> value {
  define rest = cond(length(list) == 0,
[],
    (thread() -> value {
      value = tmap(list[1:], functor);
    })());
    value = functor(list[0]) + rest;
};
```

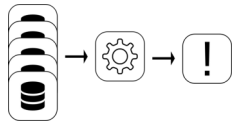
Concurrent Reduce

- Split the dataset and apply the functor one more time

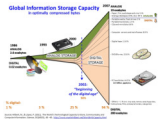
```
function treduce(list, functor) -> value {
  cond(length(list) == 2,
    value = functor(list[0], list[1]),
    {
      define threduce = thread(list) -> value
      {
        value = treduce(list, functor);
      };
      define half = length(list) / 2;
      define v1 = threduce(list[0:half - 1])
      define v2 = threduce(list[half:length(list)]);
      value = functor(v1, v2);
    });
};
```


Big Data Processing (1/2)

- MapReduce is a framework for batch processing of big data.
  - **Batch processing**:



  - **Big data**:

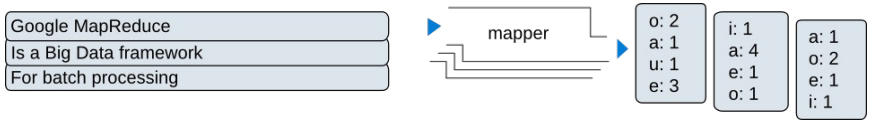


Big Data Processing (2/2)

- The MapReduce idea:
  - Data sets are too big to be analyzed by one machine
  - Using multiple machines has the same complications, regardless of the application/ analysis
  - Pure functions enable an abstraction barrier between data processing logic and coordinating a distributed application

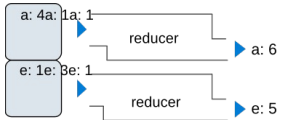
MapReduce Evaluation Model (1/2)

- **Map phase**: Apply a mapper function to all inputs, emitting intermediate key-value pairs
  - The mapper takes an iterable value containing inputs, such as lines of text
  - The mapper yields zero or more key-value pairs for each input

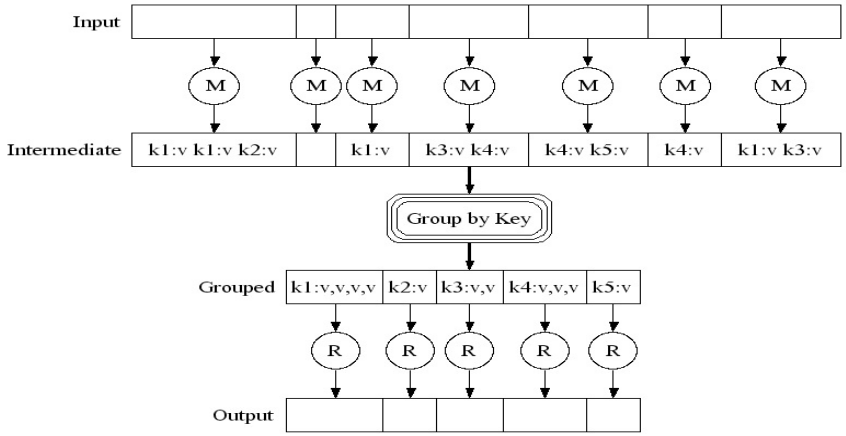


MapReduce Evaluation Model (2/2)

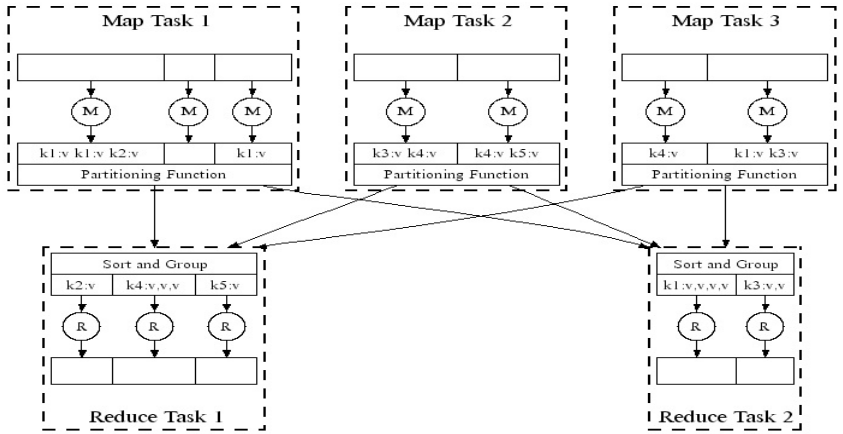
- **Reduce phase**: For each intermediate key, apply a reducer function to accumulate all values associated with that key
  - The reducer takes an iterable value containing intermediate key-value pairs
  - All pairs with the same key appear consecutively
  - The reducer yields zero or more values, each associated with that intermediate key



MapReduce Execution Model (1/2)



MapReduce Execution Model (2/2)



MapReduce example

- From a 1.1 billion people database (facebook?), we want to know the average number of friends per age
- In SQL:

```
SELECT age, AVG(friends) FROM
users GROUP BY age
```

- In MapReduce:
  - the total set of users in splitted different users_set = { user... }

```
function age_to_friends(user)
{
  send(user.age,
user.friends.size);
}
```

  - The keys are shuffled and assigned to reducers

```
function average_friends(age,
friends_counts):
{
  var r = reduce(friends_counts, 0,
      function(friends_count, s) -> r
        {
          r = friends_count.size + s
        });
  send(age, r / friends.size);
}
```

MapReduce Assumptions

- Constraints on the mapper and reducer:
  - The mapper must be equivalent to applying a deterministic pure function to each input independently
  - The reducer must be equivalent to applying a deterministic pure function to the sequence of values for each key
- Benefits of functional programming:
  - When a program contains only pure functions, call expressions can be evaluated in any order, lazily, and in parallel
- In MapReduce, these functional programming ideas allow:
  - Consistent results, however computation is partitioned
  - Re-computation and caching of results, as needed

MapReduce Benefits

- Fault tolerance: A machine or hard drive might crash
  - The MapReduce framework automatically re-runs failed tasks
- Speed: Some machine might be slow because it's overloaded
  - The framework can run multiple copies of a task and keep the result of the one that finishes first
- Network locality: Data transfer is
  - The framework tries to schedule map tasks on the machines that hold the data to be processed

Stream

Stream (1/2)

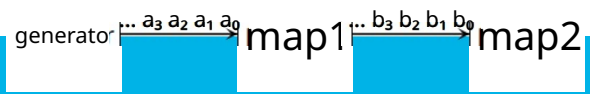
- So far, wait for the end of the execution of a thread to do the next computation

```
define a = generator(X);
define b = map1(a, Y);
define c = map2(b, Z);
...
```

- No reason to wait for a to be finished to start on b (and b for c...)

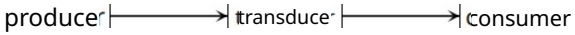
Stream (2/2)

- A stream is a sequence of message
- A stream is first-in-first-out (FIFO)
- The producer augments the stream with new messages, the consumer reads the messages, one by one



Stream communication

- **Producer**, that produces incrementally the elements
- **Transducer(s)**, that transforms the elements of the stream
- **Consumer**, that accumulate the results



Producer Pattern

```
thread producer(State, More, Produce, Transform) -> ret
{
  define s = stream();
  ret = s.ouput();
  function p(State)
  {
    cond(More(State), {
      s.send(Produce(State));
      p(Transform(State));
  };
  p(State);
}
```


Producer Pattern: Example

```
define p = generate([0, N],
  function(State)
    { return(State[0] <
State[1]); }, // More
  function(State)
    { return(State[0]); }, //
Produce
  function(State)
    { return([State[0] + 1,
State[1]]); }); // Transform
```

Consumer Pattern

```
thread consumer(State, Input, Final, Consume) -> ret
{
  define r = Input.reader();
  function c(State)
  {
    cond(r.wait(),
      c(Consume(r.next(), State)),
      ret = Final(State));
  };
  c(State);
}
```


Consumer Pattern: Example

```
define c = consume(0, I,
  function(State) {
return(State); },
  function(Next, State) {
return(State + Next); });
```


Transducer Pattern

```
thread transducer(State, Input, More, Produce, Transform) -> ret
{
  define s = stream();
  ret = s.ouput();
  define r = Input.reader();
  function t(State)
  {
    cond(r.wait(), {
      var nState = Transform(r.next(), State);
      cond(not More or More(State), {
        s.send(Produce(nState));
        t(nState)} ) });
  };
  t(State);
}
```


Transducer Pattern: Examples

```
define t1 = transducer(null, I, null,
  function(State) { return 2 * State; },
  function(Next, State) { return Next;
});
define t2 = transducer(null, I,
  function(State) { return State < 5; },
  function(State) { return State; },
  function(Next, State) { return Next;
});
```

Limitation of eager stream processing streams

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

Solutions

- Three
  1. Play with the speed of the different threads, i.e. play with the scheduler to make the producer slower
  2. Create a bounded buffer, say of size N, so that the producer waits automatically when the buffer is full
  3. Use demand-driven approach, where the consumer activates the producer when it need a new element (Lazy evaluation)
- The last two approaches introduce the notion of flow-control between concurrent activities (very common)

Applications

- Image, Video, Sound processing
- Large numerical computation, for instance, prime number computation

Stream and purity

- The full stream chain is definitely pure

```
function my_stream(...) -> ret
{
  define p1 = producer(...);
  define t1 = transducer(..., p1, ...);
  define t2 = transducer(..., t1, ...);
  ret = consumer(..., t2, ...);
}
```

- No-side effects, fully deterministic
- But:

```
thread f(...) -> ret
{
  var s = stream();
  ret = s.ouput();
  s.send(...); // side-effect?
}
```

Conclusion

- Data-driven concurrency
  - Dataflow variables
  - Implicit synchronisation
- Lazy execution
- Stream