TDDA69 Data and Program Structure
Message Passing Concurrency

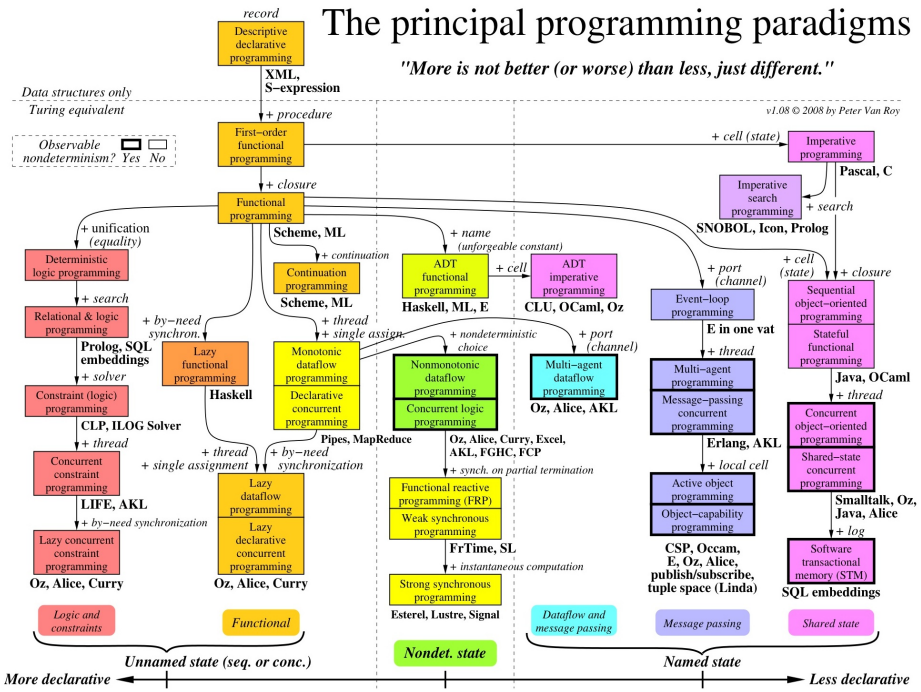
Cyrille Berger

LINKÖPING UNIVERSITY

Lecture content

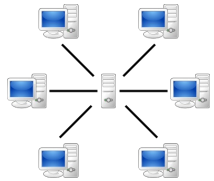
- Message passing concurrency
- Reasonning

Lectures



Client-Server Architectures

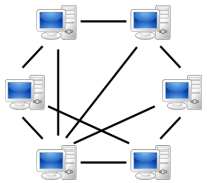
- Server provides some services
  - receives message and replies to them



  - example: web server, mail server, ...
- Clients know address of server and use service by sending messages
- Server and client run independently

Peer-to-Peer Architectures

- Similar to Client-
  - every client is also a server



  - communicate by sending messages to each other
- We call all these guys (client, server, peer): **agent**
- In the course book this is called *port object*

Common Features

- Agents
  - have identity *mail address*
  - receive messages *mailbox*
  - process messages *ordered mailbox*
  - reply to messages *pre-addressed return letter*
- Now how to cast into a programming language model?

Message Sending

- Message *data structure*
- Address *port*
- Mailbox *stream of messages*
- Reply *dataflow variable in message*

Port

- In KL we define port as:
```
port(stream);
```
- To create a port
```
define s = stream()
p = port(s)
```
- To send a message on port
```
define answer = p.send(...);
```
- Messages are received using the stream
```
s.next()
```


Receiver Pattern

```
thread receiver(State, Answer, Transform) -> ret
{
  define s = stream()
  ret = port(s)
  define reader = s.reader()
  function p(State)
  {
    cond(reader.wait(), {
      define msg = reader.next()
      s.send(Answer(State, msg))
      p(Transform(State, msg))
  }
  p(State)
}
```


Receiver Pattern: Example

```
define add_server = receiver([],
  function (state, msg) -> ret{
    ret = msg.a + msg.b
  },
  function(state, msg) -> ret
  { ret = state })
add_server.send({a: 1, b: 2})
```


Asynchronous call

- Send is blocking:
```
define answer = p.send(...);
```
- Asynchronous:
```
define answer = (thread() -> ret
{
  ret = p.send(...);
})();
```

Threaded Receiver Pattern

```
thread threaded_receiver(State, Answer, Transform) ->
ret
{
  define s = stream()
  ret = port(s)
  define reader = s.reader()
  function p()
  {
    cond(reader.wait(), {
      reader.next()
      s.send(receiver(State, Answer, Transform))
      p()
  }
  p()
}
```


Distribution

- Transparent distribution

  ```
  define r = receiver(...)
  ```

  then the receiver can be sent automatically send to an available computing unit by the interpreter
- For TCP communication

  ```
  define p = tcp_client('example.com:
  2314');
  define s = tcp_stream(2314)
  define p = port(s)
  ```


Reasoning

- The receiver or threader_receiver pattern are declarative
- As long as Answer and Transform are declarative
  - Assuming that message are received in a given order, we can use declarative reasoning
- The difficulty is in verifying the order of reception of messages

Message-Passing Concurrency

- Ports for message handling
- Client-Server, Peer-Peer communication
- Introduces non-determinism

Conclusion

- Data-driven concurrency
  - Dataflow variables
  - Implicit synchronisation
  - Lazy execution
  - Stream
- Message-Passing Concurrency