Programming in the Internet Age
–
Challenges and New Directions

Martin Odersky

EPFL

Global Computing

- Claim: *Web-based applications* represent a major paradigm shift in computing.
  - 1960's: *central* computing
  - 1980's: *local* computing
  - 2K's: *global* computing
- Global computing:
  - Computation is distributed between different sites in the internet.
  - Goal is cooperative behavior under a wide range of conditions.
- Examples:
  - internet auctions (e.g. Ebay),
  - travel reservation (e.g. Expedia),
  - load balancing (GRID),
  - group collaboration (e.g. Groove)
  - peer-to-peer information sharing.

New Challenges

Internet-wide distributed computing poses a series of challenges that are difficult to meet. For instance:

- Participants in a computation (sites) may fail.
- Sites may join or leave the system at unpredictable times (e.g. P2P)
- Message delays may be unbounded.
- Failure detection is approximate.
- Transactional semantics needs to be maintained.

Our (only?) weapons: immutability, larger granularity.

What does it mean for programming languages?

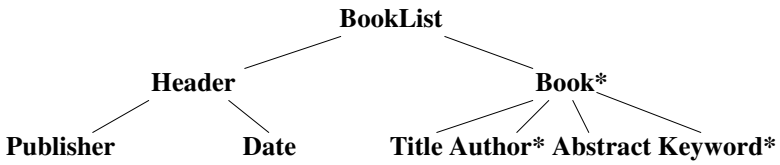
- The last shift from central to local computing triggered a change in programming language paradigms from structured to object-oriented programming.
  - Issue at the time: extensible frameworks for graphical user interfaces.
  - Extensibility with new data types requires dynamic binding.
  - This form of extensibility is predominant in
    - Simulation (hence, Simula)
    - Graphic User Interfaces (hence, Smalltalk, etc)
- Claim: the current shift from local to global computing is also likely to change programming paradigms.
  - New issues:
    Asynchronous communication,
    sharing information between sites,
    processing XML documents.

Role of a Program

- In the local computing setting
  - Programs are objects.
  - Communication is method invocation.
  - Parameters are simple values or object references.
- In the global computing setting
  - Programs are still objects, but:
  - Communication is by asynchronous message sends.
  - Parameters are immutable structured documents (e.g. written in XML).
  - Object references are problematic.

Representing XML Documents

On an abstract level, an XML documents is simply a tree.



- We can expect standard software to convert between external documents and trees.
- Trees are *pure data*; no methods are attached to nodes.
- Trees should reflect the application domain, rather than being a generic "one size fits all" such as DOM.
- This means we want different kind of data types for different kinds of tree nodes.

Parsing XML Trees

- How can trees be decomposed?
- In an object-oriented language:
  - Type test and type casts – ugly and inefficient.

```
if (node instanceof Header) {
   Header header = (Header)node;
   Publisher pub = (Publisher)header;
} else if (node instanceof Book) ...
```

  - Visitors – heavyweight, hard to extend.

```
node.visit(new Visitor() {
   void visitHeader() { ... }
   void visit Book() { ... }
}
```

- In a functional language:
  - Recursive pattern match over trees.
  - Problem again: extensibility.

The State of the Art

- Current Practice: Use patchwork of different languages for web applications. E.g..
  - C++ for the back-office,
  - C#/Java for transactions and communication infrastructure,
  - XQuery for database queries,
  - XSLT for XML document transformations.
- Lots of little (or not so little) languages, each reasonably good at their job.
- But: bad things happen at cross-language interfaces.
  - Mismatches in operational semantics.
  - No static type system to control cross-language interfaces.

A Case for Multi-Paradigm Languages?

- Instead of many different languages maybe one should use a multi-paradigm language?
- Examples:
  - Ada 95 (Imperative + OO)
  - OCaml (Functional + OO)
  - CLOS Common Lisp (Functional + OO)
  - Oz (Logic + Functional + OO)
  - ...
- Problems: Size of resulting language, semantic gaps at cross paradigm interfaces.

Scalable Languages

- Goal: develop scalable languages which can grow with the needs of their users.
  - Simple example: Add a type for complex numbers.
  - More complicated examples:
    add database querying,
    exception handling,
    process communication by first-class messages.
- Observation: scalable languages can be developed from a synthesis of object-oriented and functional programming.
- Recently, progress has been made on several issues to obtain sound foundations for such a synthesis.
  - Operational calculi:
    Join calculus, functional nets, localized pi.
  - Models of classes and objects
    (e.g. FOOL workshops)
  - Type systems:
    e.g. Objective CAML, FGJ, gbeta, nested types.

Operational Calculus: Functional Nets

- Functional nets arise out of a fusion of key ideas of functional programming and Petri nets.
- Functional programming: Rewrite-based semantics with function application as the fundamental computation step.
- Petri nets: Synchronization by waiting until all of a given set of inputs is present, where in our case

  input = function application.
- A functional net is a concurrent, higher-order functional program with a Petri-net style synchronization mechanism.
- Theoretical foundation: Join calculus.

Functions and Concurrency

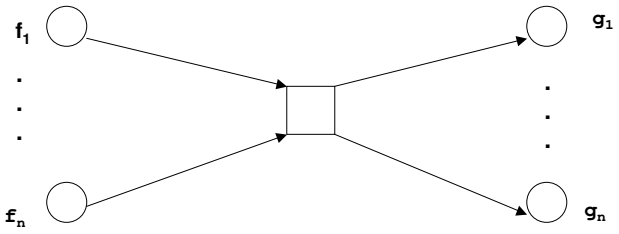
- Principle :
  - Function calls model *events*.
  - & means *conjunction* of events.
  - = means left-to-right rewriting.
  - & can appear on the right hand side of a = (*fork*) as well as on the left hand side (*join*).
- Analogy to Petri-Nets :

  call ≈ place

  equation ≈ transition

$f_1(x_1)$ & ... & $f_m(x_m)$ => $g_1(y_1)$ & ... & $g_n(y_n)$

corresponds to



- Functional Nets are more powerful:
  - parameters,
  - nested definitions,
  - higher order.

Example : One-Place Buffer

Functions : put, get (external)
empty, full (internal)

Definitions :

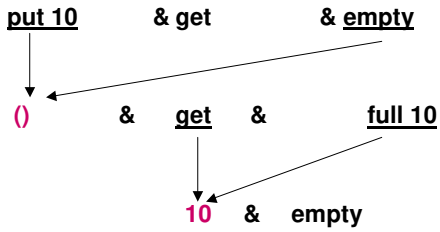
put(x) & empty => () & full(x)
get & full(x) => x & empty

Usage :

val x = get ; put (sqrt(x))

« Chemical Abstract Machine » Semantics

- A set of calls which matches the left-hand side of a rewrite rule is replaced by the rule's right-hand side (after formal parameters are replaced by actual parameters).
- Calls which do not match a left-hand side block until they form part of a set which does match.
- Example:



Type Systems – Bridging the Culture Clash

- Typical functional languages have
  - structural type systems,
  - powerful type inference methods a la Hindley/Milner,
  - powerful abstraction capabilities through abstract types
- Typcial object-oriented languages have
  - nominal type systems,
  - no type inference,
  - powerful composition capabilities through first-class objects and unrestricted recursion.
- Challenge: Combine advantages of both in one system.
  - Formalization nominal types.
  - Local type inference.
- Both were first pioneered in GJ, need to be extended now, in particular to abstract types.
- Candidate type system: Name-dependent types.

Language Implementations

- Claim: Extensibility stems from a unification (rather than addition) of features.
- Our programming language Scala has extensibility as foremost design goal (Scala = "Scalable language").
- Scala is a statically typed, object oriented language which adapts many elements found in functional languages.
- Main elements:
  - (Mixin-)classes,
  - first-class functions,
  - first-class extensible pattern matching.
- Seamless interoperability with Java and .NET is a hard requirement if the language is supposed to be useful.

Example: Auction Service

- Consider a class of auction objects.
- Auctions receive and send messages from clients (sellers and bidders)
- Need to coordinate bids, inform clients.
- Messages are XML documents, converted internally to application-specific trees.
- Messages are represented by classes like these:

```
class Message;
case class Offer(bid: Int, self: Actor) extends Message;
case class BestOffer extends Message;
...
```

A Class for Auctions

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor with {
  val increment = 10;
  def run = {
    var maxBid = minBid – increment;
    var maxBidder: Actor = null;
    while (True) {
      receiveWithin((closing – Date.currentDate).msec) {
        case Offer(bid, client) =>
          if (bid >= maxBid + increment) {
            if (maxBidder != null) maxBidder send BeatenOffer(bid);
            maxBid = bid ; maxBidder = client;
            client send BestOffer;
          } else {
            client send BeatenOffer(maxBid);
          }
```

- reserved words in **bold**; everything else is defined by user or libraries.

Auction Service ctd

```
        case Inquire(client) =>
          client send Status(maxBid + increment, closing);
        case TIMEOUT =>
          if maxBidder != null then {
            val reply = Sale(seller, maxBidder);
            maxBidder send reply;
            seller send reply;
          } else {
            seller send NoSale;
          }
        } // end receiveWithin
      } // end while
    } // end run
} // end Auction
```

Abstraction Building

- This looks somewhat like a specialized language for web applications (with process communication operations in the style of Erlang).
  - Structured messages.
  - Flexible syntax for sending and receiving messages.
  - Timeouts are built in.
- But the keywords are not where one expects them to be!
- In fact, almost all abstractions used in the example are library abstractions, so they can be defined and adapted by users.
- Idea: Provide flexible schemes for *abstraction* and *composition* rather than the primitives themselves.

Bridging the Gaps

We now see three examples of where unification of constructs gives interesting results:

- Data Variation
- Functions and Queries
- Functons and Objects

Bridging the Gaps (1): Data Variation

- Functional languages express variation using algebraic data types and pattern matching.

```
data  Message = Offer(Int, Actor) | Inquire(actor) | ...
case (message) of
   Offer(x, client) => ...
   Inquire(client) =>
```

- Advantage: It is easy to define new *processors*.
- Object-oriented languages express variation using class hierarchies.

```
class Message
class Offer extends Message { ... }
class Inquire extends Message { ... }
```

- Advantage: It is easy to define new *variants*.

Case Classes

- Like other object-oriented languages, Scala supports variation through class hierarchies.
- But it allows to *reverse* the construction process through case classes and pattern matching.

```
class Message;
case class Offer(bid: Int, client: Actor) extends Message;
case class Inquire(client: Actor) extends Message;
...
message match {
   case Offer(b, c) =>
   case Inquire(c) =>
}
```

- We thus get extensibility in both dimensions: it is easy to add new variants *and* new processors (this is essential for handling XML trees).

Bridging the Gaps (2): Functions and Queries

- Database queries are usually handled by special languages.
- Problem: Difficulty in interfacing when the query is made by a program rather than a user.
- Here's a query in Scala:

```
for (val book <- books;
     book.title contains "XML";
     val "Ullmann" <- book.authors) yield book
```

- Queries like this are mapped by the Scala compiler into applications of three higher-order functions: map, flatMap and filter.

- For instance, here's the translation of the previous query:

```
books
  .filter (book => book.title contains "XML")
  .flatMap (book =>
     book.authors
       .filter (a => a == "Ullmann")
       .map (a => book))
```

- The mapping relies only on the existence of functions map, flatMap and filter as methods of the "queried" type.

⇒ The for-construct can equally well applied in other contexts, such as combinatorial search, matrix algebra, etc.

Example: Here's the central function of the 8-queens problem:

```
def placeQueens(row: Int): List[List[Int]] =
  for (val placement <- placeQueens(row – 1);
       val column <- 1 to n;
       isSafe(column, placement, 1)) yield column :: placement;
```

map, flatMap and filter need to be defined as methods in class List.

Bridging the Gaps(3): Functions and Objects

- Scala is a pure object-oriented language: Every value is an object.
- Scala is a functional language: Functions are "first-class" values.
- Hence, functions in Scala are objects.
- They are instances of class

```
class Function[a,b] {
  def apply(x: a): b
}
```

- But what then is the status of the apply method in class Function?
- Methods are *not* functions. But a method is implicitly converted to a function when it is not immediately applied to its arguments.

Specialized Functions

- Since functions are non-final classes, they can be specialized.
- Example 1:

```
class Array[a] extends Function[Int, a] with {
  def apply: a
  def update(i: Int, x: a)
  def length: Int
}
```

- Usage:

```
a(i) = a(i-1) + a(i+1)
```

- The compiler translates this into calls of **apply** and **update**.
- Analogous for hashtables, association lists, etc.

Partial Functions

- Another useful specialization of the function type is to partial functions.
- Partial functions have an isDefinedFor method, which is used to test whether the function is defined for a given argument.

```
class PartialFunction[a,b] extends Function[a,b] with {
  def apply(x: a): b
  def isDefinedFor(x: a): Boolean
}
```

- Pattern matching case expressions are mapped by the compiler to partial functions when the context requires it.
- E.g. `{ case Offer(b, c) => b`
  `case Inquire(c) => -1 }` is of type `PartialFunction[Message, Int]`.

Application: Exception Handling

- Partial functions allow the definition of try-catch in the library.
- Example usage:

```
try {
  val data = f.read(); process(data)
} except {
  case ex: IOException => out.println(ex.getMessage)
  case ex: ClassCastException => throw (new InternalError)
}
```

- try is a function
  - which takes a block as argument
  - and which returns an object that offers an except method
- The except method takes a partial function as argument.
- The typed variable pattern ex: IOException matches any subtype of IOException and binds ex to it.

Application: Process Communication

- Message spaces are a high-level communication primitive.
- Similar to tuple-spaces, but: they are typed, and there can be many of them.
- Operations:
  - send a message,
  - retrieve a message matching a pattern.
- Signature:

```
class Message;
case class TIMEOUT extends Message;
class MessageSpace with {
  def send(msg: Message): Unit;
  def receive[a](f: PartialFunction[Message, a]): a;
  def receiveWithin[a](msec: Long)(f: PartialFunction[Message, a]): a;
}
```

Erlang-Style Processes

- An Actor is simply obtained by a mixin composition of a thread and a message space.

```
abstract class Actor extends Thread with MessageSpace;
```

- This gives a process model in the style of Erlang.
- Properties:
  - Scalable to many threads, many message kinds, structured messages.
  - Processes have only a single channel for incoming messages, hence it is easy to compose processes, e.g. forwarding, filtering, logging.
  - Thread-safety can be ensured by restricting sharing to messages, and by passing only immutable data.
  - Required: (mixin-)classes, (partial-)functions, pattern matching.
- But where Erlang's process model is fixed, Scala's is a library abstraction.

Auctions Revisited

- The Internet auction class is now revealed as an application of the actor abstraction:

```
class Auction(seller: Actor, minBid: Int, closing: Date) extends Actor with {
  val increment = 10;
  def run = {
    var maxBid = minBid – increment;
    var maxBidder: Actor = null;
    while (True) {
      receiveWithin((closing – Date.currentDate).msec) {
        case Offer(bid, client) =>
          if (bid >= maxBid + increment) {
            if (maxBidder != null) maxBidder send BeatenOffer(bid);
            maxBid = bid ; maxBidder = client;
            client send BestOffer;
          } else {
            client send BeatenOffer(maxBid);
          }
        ...
```

History

- Scala evolved from our work on functional-nets
- First goal was to build a minimal programming language based on a universal calculus of computation.
- We relied on known encodings to obtain pattern matching and classes.
- User experience taught us that encodings work fine, but only for experts.
- Scala tries to have wider appeal by including more constructs in the base language.
- New question: How to combine object-oriented and functional constructs and how to unify them where possible.
- This has proved to be much more interesting than expected!

Conclusion

- Scala pushes the state of the envelope in what abstractions can be built by library designers.
- It makes it possible to express many domain specific languages as libraries.
- Of course, this expressiveness needs to be used intelligently and responsibly – it is just as easy to build a bad library than it is to build a bad language.
- But, given the choice, I prefer a bad library because it is easier changed or replaced.

Project Status

- Compiler for a language subset exists.
- Used in mandatory course for 2nd year students at EPFL.
- Output language is Java bytecodes (.NET version is under development).
- Compiler maps as directly as possible to underlying architecture.

⇒ Easy interoperability by cross-language method invocation and inheritance.

- Started cooperation with Benjamin Pierce on integration of XDuce's successor Xstatic and Scala.
- To find out more: lampwww.epfl.ch/~odersky/scala