Trends in Languages
2009 Edition

Markus Völter
Independent/itemis
voelter@acm.org
www.voelter.de

About me

- Independent Consultant for itemis
- Based out of Stuttgart, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Product Line Engineering

Markus Völter
voelter@acm.org
www.voelter.de

Founder and Editor of
Software Engineering Radio
the Podcast for Professional Developers
http://se-radio.net
Trends in Language 2009

CONTENTS

• Intro and Overview
• Typing
• OO +/vs. Functional
• Metaprogramming
• DSLs
• Concurrency
• Platforms
• Tools
• Summary
Why this talk?

• Language World **is changing**
  • Mainstream Languages evolve (Java, C#)
  • Diverisfication: Ruby, Erlang, Scala, Groovy, ...

• I want to illustrate interesting **trends**

• Explain some of the **controversy** and **backgrounds**.

---

CONTENTS

• Intro and Overview

• **Typing**
  • OO +/-vs. Functional
  • Metaprogramming
  • DSLs
  • Concurrency
  • Platforms
  • Tools
  • Summary
Different kinds of typing

- Does a language have types at all?
- Are those typed checked at all?
  - No: Bad!
  - Yes: ...
- If so, when?
  - Runtime: Duck Typing
  - Compiler: Static Typing

Duck Typing

- A form of Dynamic Typing
  "if it walks like a duck and quacks like a duck, I would call it a duck"
  - where not the declared type is relevant
  - but the ability at runtime to handle messages/method calls
- A handler for a message (method implementation) can be
  - Defined by its type
  - Be object-specific
  - Added at runtime via meta programming
- A predefined callback ("doesNotUnderstand") is invoked in case a message cannot be handled.
- Examples: Smalltalk, Ruby
**Structural Types: Duck Typing for Static Languages**

- **Compiler** checks, whether something can satisfy context requirements.
  - Formal type is not relevant

- **Example I:** C++ Templates

- **Example II:** Scala

  ```scala
  class Person(name: String) {
    def getName(): String = name
  }
  def printName(o: { def getName(): String }) {
    print(o.getName)
  }
  printName( new Person("markus") ) // prints "markus"
  ```

**Type Inference: Omit derivable types**

- **Compiler Smarts:** You only have to write down those types the compiler cannot derive from the context

- **Example:** (Hypothetical) Java

  ```java
  // valid Java
  Map<String, MyType> m = new HashMap<String, MyType>();
  // Hypothetical Java with Type inference
  var m = new HashMap<String, MyType>();
  ```

- **Example II:** C# 3.0, LINQ

  ```csharp
  Address[] addresses = ...
  var res = from a in addresses
              select new { name = a.name(),
                           tel = a.telephoneNo() };
  foreach (var r in res) {
    Console.WriteLine("Name: {0}, Num: {1}", r.name, r.tel);
  }
  ```
Optional/Pluggable Types

- Fundamentally, types are checked **at runtime**

- However, optionally one can add **type annotations** that are checked statically by a type checker
  - Why should types be treated differently than any other form of meta data that is worth checking? Concurrency, Timing, ...

- Types are just **meta data** for which a checker is available

- Several **different kinds of meta data** (i.e. type systems) can be optionally overlayed over the same program

- Like **static analysis** tools that rely on custom annotations in Java (@CanBeNull)

- **Example**: Newspeak / Gilad Braha

## CONTENTS

- Intro and Overview
- Typing
- **OO +/- vs. Functional**
- Metaprogramming
- DSLs
- Concurrency
- Platforms
- Tools
- Summary
OO and Functional

- OO is clearly the **mainstream**.
- That is changing (very) slowly ... especially **functional programming** is taking up speed.
- What is functional programming (as in Erlang, Lisp, F#)
  - Function Signatures are **types**
  - Function **Literals** are available (lambda expressions)
  - Functions are **values**: assignable to variables and parameters → Higher Order Functions
- You can find **elements** of this in Ruby, Groovy, C# 3 and Scala
- Scala’s primary goal is to unify OO and functional
- (also: **side-effect free**; important later wrt concurrency)

From Primitive To Workable

- **Primitive** functional programming can be done with
  - Function pointers (as in C/C++)
  - Delegates (C# < 3)
  - Command Pattern/Inner Classes in Java
- Better solution: **Closures**
  (aka lambda expressions, blocks, anonymous functions)

```
[1,2,3,4,5,6].each { |element| puts (element * 2) }  # Ruby
```

- **Anonymous Functions** (Function Literals)

```
x: Int => x + 1  # Scala
```
Higher Order Functions

- **Function Signatures** (Function Types)
  - \(\text{Int} \Rightarrow \text{Int} \quad // \quad \text{Int Parameter, Return Type Int}\)
  - \((\text{Int}, \text{Int}) \Rightarrow \text{String} \quad // \quad \text{Two Int Parameter, returns String}\)

- Function Signatures/Types are important for **Higher Order Functions**:
  - Functions that take other functions as arguments
  - ... or return them

  ```scala
  def apply(f: Int => String, v: Int) => f(v)
  ```

Currying

- Functions called with fewer arguments than they formally expect **return new functions** with the given parameters bound

  ```fsharp
  > let add a b = a + b;;
  val add : int -> int -> int
  > let add10 = add 10;;
  val add10 : int -> int
  ```

- Name „currying“ based on **Haskell Curry**
  (yes, his first name was used for the Haskell language)
Pattern Matching

• Checking for and extracting the structure of data

```fsharp
let t = (42, "text");
val tuple : int * string

let (num, str) = tuple;
val num : int // 42
val str : string // "text"
```

• Especially useful for discriminated unions (F#) or case classes (Scala) to decompose tree structures (e.g. expression trees)

```fsharp
type Expr =
| Op of string * Expr * Expr
| Var of string
| Const of int;

let expr = Op("+", Var "x", Const 10);
val expr : Expr
```

Pattern Matching II

• A simple evaluator for the expressions using pattern matching

```fsharp
let rec eval x = match x with
| Op(op, l, r) -> let (lv, rv) = (eval l, eval r)
  if (op = "+") then lv + rv
  elseif (op = "-") then lv - rv
  else failwith "Unknown operator!"
| Var(var) -> getFromSymbolTable var
| Const(n) -> n;

val eval : Expr -> int
```

• Patterns can deconstruct the composition structure of complex data structures and assign local variables to the parts of the data structure.
What is Metaprogramming?

• A program can inspect and modify itself or other programs.

• Not a new concept: Lisp, CLOS
  • But returning to fame these days...

• Two different flavours:
  • Static/Compile Time metaprogramming: handled by compiler
  • Dynamic metaprogramming: done at runtime
    (fits well with Duck Typing ... you can call what’s there)

• Static Meta Programming is a relative niche concept
  (aka hygienic macro system)
  • C++ Template Metaprogramming (aargh!)
  • Template Haskell
  • Converge
  • Boo
Dynamic Metaprogramming

- Is available in many dynamic OO languages, such as Smalltalk, Ruby, Groovy

- Dynamically add a new method to a class:

```ruby
class SomeClass
  define_method("foo") { puts "foo" }
End
SomeClass.new.foo // prints "foo"
```

- What happens in Duck languages, if you call a method that's not available? Remember, no compiler type check!

```groovy
class Sammler {
  def data = []
  def propertyMissing = {String name, value-> data [name] = value }
  def propertyMissing = {String name-> data [name] }
}
def s = new Sammler()
s.name = "Voelter"
s.vorname = "Markus"
s.name // is "Voelter"
```

Meta Object Protocols

- MOPs support „overwriting“ the interpreter typically via the concept of meta classes.

- Here we overwrite what it means to call a method:

```groovy
class LoggingClass {
  def invokeMethod(String name, args) {
    println "just executing "+name
    // execute original method definition
  }
}
```

- Yes, this looks like the AOP standard example 😊

- In fact, AO has evolved from MOPs (in CLOS)

- And now we’re back to MOPs as a way for „simple AO“… strange world …
What are DSLs?

A DSL is a **focused, processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- Domain can be **business** or **technical** (such as architecture)
- The “program” needs to be **precise** and **processable**, but not necessarily **executable**.
  - Also called **model** or **specification**
Internal DSLs vs. External DSLs

- **Internal DSLs** are defined as part of a host language.
  - DSL „program“ is **embedded** in a host language program
  - It is typically **interpreted** by facilities in the host language/program (→ metaprogramming)
  - DoF for syntax customization is **limited by host language**
    - Only useful in languages with a **flexible syntax** (such as Ruby) or no syntax (Lisp ☺)

- **External DSLs** are defined independent of any programming language
  - A program **stands on its own**.
  - It is either **interpreted** by a custom-build interpreter, or **translated** into executable code
  - DoF for syntax customization **only limited by custom editor** (i.e. not really limited at all: graphical, textual, tables, combinations of those...)

Dynamic Internal DSL Examples: Ruby

- **Ruby** is currently the **most suitable language** for internal DSLs.

```
class Person < ActiveRecord::Base
  has_one :adress
  has_many :telecontact
end

class Address < ActiveRecord::Base
end
```

- **has_one** and **has_many** are actually **invocations of class methods** of the **ActiveRecord::Base super class.**

- Alternative Syntax:

```
has_one(„adress“)
```

- The original notation is an example of **Ruby’s flexible syntax** (optional parens, symbols)
Dynamic Internal DSL Examples: Ruby II

- The `has_one` and `has_many` invocations dynamically create accessors for properties of the same name:

```ruby
p = Person.new
a = Address.new
p.adress = a
p.adress == a
```

- The methods are implemented via meta programming.

- They do all kinds of magic wrt. to the database backend used in Rails.

Static Internal DSL Examples: Scala

- The following uses `loop/unless` as if it were a Scala language feature (which it is not!)

```scala
var i = 10;
loop {
  Console.println("i = " + i)
  i = i-1
} unless (i == 0)
```

- In fact, it is implemented as a library relying on automatic closure construction and the use of methods in operator notation.

```scala
def loop(body: => Unit): LoopUnlessCond = new LoopUnlessCond(body);

private class LoopUnlessCond(body: => Unit) {
  def unless(cond: => Boolean): Unit = {
    body
    if (!cond) unless(cond);
  }
}```
Static Internal DSL Examples: Boo

- Boo has a full **hygienic macro system** (open compiler)

```csharp
public interface ITransactionable:
    def Dispose(): pass
    def Commit(): pass
    def Rollback(): pass
```

```csharp
macro transaction:
    return [|
    tx as ITransactionable = $(transaction.Arguments[0])
    try:
        $(transaction.Body)
        tx.Commit()
    except:
        tx.Rollback()
        raise
    finally:
        tx.Dispose()
    |]
```

- Use it like **native language syntax!**

```csharp
transaction GetNewDatabaseTransaction():
    DoSomethingWithTheDatabase()
```

Static Internal DSL Examples: Boo II

- See how the **Expression** type is used to **pass in AST/syntax elements** (in this case, an expression)

```csharp
[ensure(name is not null)]
class Customer:
    name as string
    def constructor(name as string): self.name = name
    def SetName(newName as string): name = newName
```

```csharp
[AttributeUsage(AttributeTargets.Class)]
class EnsureAttribute(AbstractAstAttribute):
    expr as Expression
    def constructor(expr as Expression): self.expr = expr
    def Apply(target as Node):
        type as ClassDefinition = target
        for member in type.Members:
            method = member as Method
            block = method.Body
            method.Body = [|
                try:
                    $block
                    ensure:
                        $expr
                |].Block
```

Boo examples taken from Ayende Rahien and Oren Eini's InfoQ article *Building Domain Specific Languages on the CLR*
Language Workbenches à la Fowler

- Traditionally, the **AST (Abstract Syntax Tree)** is the result of a parsing process – ascii text is the master

- In Projectional Editors, the **tree is the master**, and the editor, as well as the (potentially) generated code follows from **projections** (i.e. model-2-model transformations)

Benefits of Projectional Editing

- Syntax can be **ambiguous**

- **Language Modularization** becomes much simpler – because see above

- Language modules can be **composed at the site of use** – no explicit grammar composition necessary.

- A much **larger range of symbols** can be used

- **Textual** can **graphical** notations can be treated similarly
  - Simplifies mixing
  - Semi-graphical notations (mathematical symbols) can be used

- **NB:** Parser technology also evolves, scannerless parsers can do **some** of the same things.
Disadvantages of Projectional Editing

- **You edit a tree** – depending on the tool this can be a pain (or at least take some getting used to)

- **Storage** based on abstract syntax (maybe XML), i.e. text-based tooling won’t work anymore.

- Everything will be **tool-dependent**... no standards as of now.

---

Long Range Vision: Modular Languages

- We won’t have **large, general purpose languages** anymore, and use **modeling** for all the stuff where the GPLs are lacking.

- Instead we will have **modular languages**,
  - with **standardized modules** for technical concerns
    - Remoting, persistence, state machines, ...
  - And the ability to **build custom modules** for our own domains

- Realistic? Absolutely.
MPS as a Language Workbench

- jetbrains.com/mps
- **Open Source**, Apache 2.0
- **Java as a Base Language** – can be extended!
- Tree editing ok (takes getting used to)

**Timeline**
- Beginning of 2009: Public Beta
- March 2009: Beta 2
- sometime in Q2/2009: 1.0 final

MPS: Java with Lock Statement

```java
public class Test extends <name> implements <name> {
  <static fields>
  <static initializers>
  private LockHelper helper = new LockHelper();
  <properties>
  public Test() {
    <no statements>
  }
  private int a() {
    int x = 0;
    int i = 0;
    lock (this.helper.getLock())
    int l = i;
    i = i + 1;
    <statements>
  }
  return 0;
  }
  <static methods>
  <static inner classifiers>
}
```

- **translated** to „normal“ Java for compilation
- ca. 15 minutes of work to extend Java with the **LockStatement**.
Trends in Languages 2009

Why?

• Systems need to **scale**: More and More machines

• **Machine performance** needs to improve: Multicore
  • Multicore system can provide **real concurrency** as opposed to „apparent“ concurrency on one core.
  • Multicore systems can only be utilized fully if the available set of cores is utilized effectively.
The role of pure functional programming

- **Pure Functional Programming** uses
  - Only functions without sideeffects
  - No shared state
  - Immutable data structures

- If you share nothing (or the shared stuff is not mutable) there’s no need for locking or other access coordination protocols → pure functional languages are a good fit

- The call graph is the only dependency structure in the system (no hidden dependencies using global/shared state)
  - makes the programs easier (or even feasible) to analyze
  - And makes parallelization simple (you can parallelize any set of sub-callgraphs)

Shared Memory Concurrency

- Mainstream languages use **shared memory**:
  - A process (address space) can host any number of thread
  - Threads can share data
  - They need to coordinate via locking

- Locking has to be implemented manually by developers via an agreed locking/coordination protocol
  - Often very complex (non-local)
  - Error prone, because there’s little language/tool support
  - Overspecification: „Acquire/Release Lock X“ vs. „Pretend this were sequential/atomic“

- Solution: **Express atomicity requirements** with language primitives as opposed to using locking protocol API → Transactional Memory
Shared Memory Concurrency: Transactional Memory

- Transactional Memory in Fortress:
  ```fortress
  atomic do
  // the stuff here is executed as if
  // there was only this thread
  end
  ```

- This formulation says nothing about specific locks and their allocation and release:
  - Less error prone
  - More potential for optimizations of compiler and RT system

- Similar in Spirit to Garbage Collection (Dan Grossman):
  - Rely on clever compiler and RT system
  - Solution might not always be optimal
  - ... but good enough in 99% of cases
  - and much less (error prone) work.

More bad overspecification

- Overspecification generally prohibits a compiler or runtime system from introducing optimizations.

- Example: Assume you want to do something for each element of a collection

- (Old) Java solution enforces total ordering. Intended?
  - Compiler cannot remove ordering
  ```java
  for (int i=0; i < data.length; i++) {
    // do a computation with data[i]
  }
  ```

- (New) Java solution: no ordering implied
  - Intent is expressed more clearly
  ```java
  foreach (DataStructure ds in data) {
    // do something with ds
  }
  ```
The default is parallel

- In Fortress, a loop is by **default parallel**
  - i.e. the compiler can distribute it to several cores

```fortress
for I <- 1:m, j <- 1:n do
  a[I,j] := b[I] * c[j]
end
```

- If you need **sequential** execution, you have to **explicitly specify** that.

```fortress
for i <- seq(1:m) do
  for j <- seq(1:n) do
    print a[i,j]
  end
end
```

- Fortress does more for concurrency:
  - it knows about **machine resources** (processors, memory)
  - **Allocates** to those resources explicitly or automatically

"Shared Memory is BAD" (Joe Armstrong)

- Some (many?) claim that the **root of all evil is shared memory** (more specifically: shared, mutable state):
  - If you **cannot modify** shared state, no need for locking
    - Fulfilled by pure functional languages
  - If you **don't even have shared state**, it's even better.
    - This leads to message-passing concurrency
    - Aka Actor Modell
  - **Erlang**: most prominent example language (these days)
    - **Functional Language**
    - Conceived of 20 years ago at Ericsson
    - Optimized for distributed, fault tolerant (telco-)systems
    - Actors/Message Passing based (called Process there 🌎)
"Shared Memory is BAD" (Joe Armstrong)

- Some (many?) claim that the root of all evil is shared memory (more specifically: shared, mutable state):
  - If you cannot modify shared state, no need for locking
    - Fulfilled by pure functional languages
  - If you don't even have shared state, it's even better.
    - This leads to message-passing concurrency
    - Aka Actor Modell
- Erlang: most prominent example language (these days)
  - Funtional Language
  - Conceived of 20 years ago at Ericsson
  - Optimized for distributed, fault tolerant (telco-)systems
  - Actors/Message Passing based (called Process there 😃)
**Actors/Message Passing in Erlang**

- The **only way to exchange information** between actors is via message passing.

  - *Spawn creates a new process* – it executes the lambda expression passed as an argument

    \[
    \text{Pid} = \text{spawn}(\text{fun}() \rightarrow \text{doSomething() end})
    \]

  - **Sending** a message (any Erlang data structure) happens via the `!` notation

    \[
    \text{Pid} ! \text{Message}
    \]

---

**Actors/Message Passing in Erlang II**

- An Actor’s **received messages** are put into a “mailbox”

  - A Unix Select-like command **receive** takes out one at a time.

  - **Pattern Matching** is used to distinguish between the different messages
    - **lower case**: constants
    - **upper case**: free variables that will be bound

    \[
    \text{loop}
    \begin{align*}
    \text{receive}
    \{\text{add}, \text{Id}, \text{Name}, \text{FirstName}\} & \rightarrow \text{ActionsToAddInformation}; \\
    \{\text{remove}, \text{Id}\} & \rightarrow \text{ActionsToRemoveItAgain}; \\
    \ldots & \\
    \text{after Time} & \rightarrow \text{TimeOutActions}
    \end{align*}
    \]

    \[
    \text{end}
    \]
Erlang-Style Message Passing in Scala

• Necessary ingredients for Actors include
  • Closures
  • Efficient Pattern Matching

• Scala has those features, too.
  • It also provides a way to define new “keywords” (receive) and operators (!)

```scala
receive {
  case Add(name, firstName) => …
  case Remove(name, firstName) => …
  case _ => loop(value)
}
```

• This piece of Scala code doesn’t just look almost like the Erlang version, it also performs similarly.

Best of Both Worlds in Singularity

• MP disadvantage: message data copying overhead

• Singularity (Sing#) solution: Best of Both Worlds
  • Use message passing semantics and APIs
  • But internally use shared memory (memory exchange)
  • Enforce this via static analysis in compiler

• Example (pseudocode)

```scala
struct MyMessage {
  // fields
}

MyMessage m = new MyMessage(…)
receiver ! m
// use static analysis here to ensure that
// no write access to m
```
Languages vs. Platforms

- Virtual Machines: Let’s have a small set of stable, fast, scalable platforms and a larger variety of languages for different tasks running on those platforms.
  - CLR has always had a clear distinction
  - JVM is getting there: JRuby, Jython, Groovy, Scala
    • `invokedynamic`, tail recursion

- The same concept applies to enterprise platforms: JEE as an „operating system“ for enterprise apps has
  - Scalability
  - Deployment
  - Manageability, Operations

- … and use different languages/frameworks on top of this „Enterprise OS“
  - This is an advantage of Groovy/Grails vs. Ruby/Rails
When defining a language, always think about tooling!

- Tooling includes
  - **editing** (coloring, code completion, refactoring, etc.)
  - (static) **analysis**

- Powerful tooling is **simpler** to build for **statically typed** languages

- However, **IDEs for dynamic languages** are feasible, too:
  - Netbeans Ruby support
  - Smalltalk Browsers

- **Metaprogramming** is simpler to do in **dynamic languages**
  - there's no tooling to be adapted with the language
  - How can the IDE know about changes to programs at RT?
  - Compile-Time meta programming does not include tooling
When defining a language, always think about tooling! II

- Internal DSLs – implemented mostly in dynamic languages – **do not provide any tool support** for the DSL
  - Main disadvantage of dynamic, internal DSLs
  - Usability for business user limited!!

- In **external DSLs** you build a **custom editor** which then typically provides the well-known IDE productivity features (to one extend or another). Examples include
  - GMF for graphical notations
  - Xtext for textual notations

- **Static Analysis** becomes a central issue for **concurrency**
  - If concurrency is supported on **language level**, more compiler/analysis support becomes available.
  - MS Singularity Project is a good example

**C O N T E N T S**

- Intro and Overview
- Typing
- OO +/-vs. Functional
- Metaprogramming
- DSLs
- Concurrency
- Platforms
- Tools
- **Summary**
Summary

- The time when only one language rules are over.
- Languages are a **topic of discussion** again
- It's about **language concepts**, not little details!
- New Buzzword: **Polyglott Programming** (new concept?)
  Build a system using several languages,
  - A robust, static, compiled languages for the **foundation**
  - The more volatile parts are done with a more productive, often dynamically typed language
  - DSLs are used for **end-user** configuration/customization
- Languages I could have talked about:
  - F# (functional), Ada 2005 (concurrency)

CONTENTS

- Intro and Overview
- Typing
- OO +/- Functional
- Metaprogramming
- DSLs
- Concurrency
- Platforms
- Tools
- Summary

THANKS!