Trends bei Sprachen

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

Bei den Programmiersprachen tut sich was: der Sprachenmarkt ist wieder in Bewegung gekommen. Das gilt zum Einen für die Weiterentwicklung der Mainstream-Sprachen Java und C#, zum Anderen aber auch generell für die Sprachvielfalt. Generell scheint die Erkenntnis, dass man auch mit noch so guten Frameworks keine mächtigen Sprachfeatures ersetzen kann wieder in den Vordergrund zu rücken. In diesem Artikel möchte ich daher einige aktuelle Trends bei Programmiersprachen beleuchten. Ziel ist es, Sie als Leser auf diese (mehr oder weniger) neuen Trends aufmerksam zu machen und ihnen einige Hintergrundinformationen zu bieten. Dabei geht es auch darum, ein bischen über den Tellerrand hinauszuschauen – selbst wenn es die betreffenden Features in der Sprache in der man arbeitet nicht gibt, so kann man doch seinen Programmierstil von diesen Features inspirieren lassen.

Hier ein kurzer Überblick über die behandelten Themen: Zunächst gehe ich auf die verschiedenen Methoden der Typisierung ein (schwach, stark, dynamisch, statisch, duck, structural, …) um diese Dauerdebatte etwas zu entschärfen. Im zweiten Abschnitt möchte ich erläutern, wie und warum Konzepte der funktionalen Programmierung derzeit Einzug in den OO Mainstream halten. Ein weiteres wichtiges Thema ist Metaprogrammierung, das inbesondere im Zusammenhang mit Ruby derzeit viel Aufmerksamkeit erfährt. Gleiches gilt für Domänenspezifische Sprachen, allerdings ist das Thema erheblich breiter als „nur" Ruby's interne DSLs. Darüber möchte ich einen kurzen Überblick geben. Das Thema Concurrency wird sich immer mehr in den Vordergrund drängen (*The free lunch is over [0]*). Können Sprachen dabei helfen, das Thema besser beherrschbar zu machen? Abschließen möchte ich mit einer kleinen Diskussion über Plattformen vs. Sprachen und über Tools zur statischen Validierung von Programmcode

Die folgende Tabelle gibt einen Überblick über die im Artikel erwähnten Sprachen, die wichtigsten Features im Zusammenhang mit dem Artikel, sowie einige Quellen um weitere Informationen über die Sprachen zu bekommen.

Sprache	Relevanz bzgl. des Artikels
Java	Beispiel einer aktuellen Sprache, die viele der hier besprochenen Features vermissen lässt.
C++	Strukturelle Typen im Zusammenhang mit Templates
Scala [2]	Type Inference, Strukturelle Typen, Funktionale Programmierung, DSLs, Concurrency
C# 3.0 [3]	Typ Inference
Ruby [5]	Duck Typing, Meta Programming, DSLs
Groovy [6]	Metaprogrammierung, DSLs
Fortress [19[	DSLs, Concurrency
Erlang [34]	Concurrency

Die Typisierungsdebatte

Seit Jahr und Tag tobt eine heiße Debatte um die Frage, ob Sprachen statisch oder dynamisch typisiert werden sollten – mit dem Unterton, dass „richtige" Sprachen statisch typisiert sind, und alles andere „Skriptsprachen" sind. Diese Klassifizierung ist eine grobe Vereinfachung – ich möchte hier daher versuchen, etwas Klarheit in die Debatte zu bringen (siehe auch [1]).

Wenn man über die Typisierung von Sprachen redet, so muss man zunächst einmal fragen, ob die Sprache überhaupt Typen kennt und diese überprüft. C zum Beispiel ist eine (zumindest potentiell) schwach typisierte Sprache, weil man mit einem *void** so ziemlich alles machen kann. Beispielsweise kann man einen String als Zahlenfolge interpretieren, wenn man über *void** castet. Das bedeutet, dass das Typsystem sozusagen ein Loch hat, durch das man den Compiler hintergehen kann. Ein Typsystem, welches es erlaubt, eine Variable als etwas zu interpretieren, was sie eigentlich nicht ist, nennt man schwach. Die Community ist sich im Wesentlichen einig, dass ein schwaches Typsystem nicht wünschenswert ist.

Im Gegensatz dazu steht die strenge Typisierung. Diese lässt sich weiter untergliedern. Findet die Typprüfung durch den Compiler statt, redet man von statischer Typisierung. Wird sie vom Laufzeitsystem erledigt, nennt man dies dynamische Typisierung. Die meisten Mainstream-Sprachen (Java, C#, C++) verwenden statische Typisierung. Dynamische Typisierung wird im Allgemeinen mit Skriptsprachen assoziiert. Man beachte, dass fast alle statisch typisierte Sprachen eine Hintertür besitzen, durch die die Typprüfung auf die Laufzeit vertagt werden kann: casting.

Hier setzt die Debatte an: „statische Typisierung ist besser, weil der Compiler mehr Fehler erkennen kann" sagen die Einen, „dynamische Typisierung ist besser, weil man damit flexibler ist (und Tests muss man ja eh schreiben)" sagen die Anderen. Meiner Meinung nach haben beide Arten ihre Vor- und Nachteile, und es kommt eben darauf an, wofür man sie einsetzt: Sicherheitskritische Software wird man vermutlich lieber mit einem statischen Typsystem, agile Webanwendungen mit einer dynamischen Sprache schreiben wollen.

Es gibt allerdings noch weitere erwähnenswerte Unterarten und Kombinationen. *Duck Typing* ist eine Art von dynamischer Typisierung, bei der es auf die aktuelle Struktur eines Objektes ankommt und nicht auf seinen deklarierten Typ (daher auch der Name: *if it walks like a duck and quacks like a duck, I would call it a duck* [43]). Man kann einem Objekt all die Nachrichten schicken, die es versteht. Dabei ist es unerheblich, ob die Nachrichten in seiner Klasse deklariert wurden, ob sie objektspezifisch sind oder per Metaprogrammierung zur Laufzeit hinzugefügt wurden. Wenn man einem Objekt eine Nachricht schickt die es nicht versteht, wird üblicherweise ein Callback aufgerufen, mit Hilfe dessen man auf „unbekannte" Nachrichten programmspezifisch reagieren kann. Smalltalk und Ruby beispielsweise verwenden ein derartiges Typsystem.

Auch wenn der Begriff Duck Typing fast ausschließlich im Kontext dynamischer Sprachen Verwendung findet, ist das Konzept streng genommen nicht auf dynamische Sprachen beschränkt: C++ Templates funktionieren auch so. Hier überprüft der Compiler bei Instanziierung einer Template ob ein Datentyp den Anforderungen strukturell genügt; es kommt nicht auf den formalen Typ an. In Scala gibt es das Konzept der strukturellen Typen [2]. Hier ein Beispiel:

Scala
```
class Person(name: String) {
  def getName(): String = name
  …
}

def printName(o: { def getName(): String }) { print(o.getName) }

printName( new Person(“markus”) )  // prints “markus”
```

Wir definieren eine Funktion *printName,* die einen Parameter *o* erwartet. *o* ist allerdings nicht mittels einer Klasse typisiert. Statt dessen muss die Klasse des Parameterobjektes eine Methode *getName(): String* besitzen. Da die Klasse Person eine solche Methode hat, können Personenobjekte an die Methode *printName* übergeben werden.

Eine weitere interessante Entwicklung im Bereich der statischen Typisierung ist *Type Inference* [7] (natürlich ist sie nicht wirklich neu, aber sie hält langsam Einzug in den Mainstream). Dabei muss man all die Typen, die der Compiler automatisch ableiten kann, nicht explizit hinschreiben. Dies wird in Scala [2] und in C# 3.0 [3] ausführlich eingesetzt. Type Inference macht ein Programm schlicht besser lesbar. Hier ein Beispiel in Java:

Java
```
// gültiges Java
Map<String, MyType> m = new HashMap<String, MyType>();
// Java, wenn es Type Inference unterstützen würde,
// hier am Beispiel von Generics
var m = new HashMap<String, MyType>();
```

Ein besonders schönes Anwendungsbeispiel für Type Inference ergibt sich im Zusammenhang mit LINQ [4], dem neuen Language Integrated Query Mechanismus in C# 3. Dabei ist es möglich, Queries gegenüber verschiedenen Datenformaten mit demselben, in die Sprache integrierten Mechanismus abzusetzen. Hier ein Beispiel mit in-memory Datenstrukturen:

C# 3.0
```
Address[] addresses = …

var res = from a in addresses
          select new {name = a.name(), tel = w.telephoneNo()};

foreach (var r in res) {
  Console.WriteLine("Name: {0}, Num: {1}", r.name, r.tel);
}
```

Das *select* Query pickt aus der Liste der *Address*-Objekte 2-Tupel der Form *{name, telefonNummer}* heraus. Genau genommen selektiert dieses Query also nicht, sondern projiziert. Dabei werden mittels *new {…}* Instanzen eines anonymen Typs erzeugt. Man beachte, dass der Rückgabewert des Queries einfach mittels *var* deklariert wird. Wir geben keinen expliziten Typ an, der Compiler kann sich errechnen, dass es sich um eine Kollektion handeln muss. Deshalb kann man im Folgenden auch mittels *foreach* darüber iterieren. Richtig interessant wird es aber im *WriteLine*-Statement. Dort greifen wir auf die Attribute *name* und *tel* des anonymen Typs zu, der der durch die Projektion hervorging,. Der Typ wird vom Compiler automatisch erzeugt – er kennt ja die Struktur. Der Entwickler muss den Namen des Typs aber nicht kennen, da er ihn nie explizit angeben muss: *Type Inference* sei Dank. Trotzdem werden sämtliche Typprüfungen statisch vom Compiler erledigt. Es handelt sich hier *nicht* um Duck Typing, auch wenn der Eindruck nahe liegt.

OO + Funktional

Über die letzten 15 bis 20 Jahre war der Mainstream klar von der Objektorientierung dominiert. Andere Paradigmen hatten einen schweren Stand. Dies ändert sich nun langsam aber sicher. Besonders die funktionale Programmierung ist auf dem Weg in den Mainstream. Funktionale Programmierung [8] ist unter anderem dadurch gekennzeichnet, dass Funktionen als zentrale Abstraktionsmechanismen verwendet werden. Dies bedeutet unter anderem, dass

- Funktionssignaturen Typen sind,
- es eine Syntax für Funktionsliterale, auch *lambda*-Expressions genannt, gibt (dazu später mehr),
- Funktionen an Variablen gebunden oder als Parameter an andere Funktionen übergeben werden können; dies resultiert in Funktionen höherer Ordnung.

Diese Charakteristiken findet man sowohl in Ruby [5], Groovy [6], C# 3.0 als auch in Scala. Eines der Hauptziele von Scala ist es eben gerade, objektorientierte und funktionale Programmierung harmonisch zu kombinieren. Pur funktionale Sprachen sind außerdem zustandslos und seiteneffektfrei. Diese Eigenschaften sind für die Diskussion hier nicht relevant, werden aber weiter unten im Zusammenhang mit Concurrency wichtig.

Ein erster, primitiver Schritt zur funktionalen Programmierung sind Funktionszeiger, wie man sie aus C, C++ (Funktionsobjekte [44]) oder C# (Delegates [45]) kennt. Die C++ STL Containerbibliotheken verwenden dieses Feature ausführlich. In gewisser Weise gibt es sie auch in Java: man muss sie mittels Interfaces und (gegebenenfalls anonymen) Klassen emulieren. Allerdings ist die resultierende Syntax sehr umständlich.

Um dieses Problem zu lösen, wird die nächste Version von Java aller Voraussicht nach Closures enthalten. Closures sind letztendlich *lamda*-Expressions bzw. anonyme Funktionen. In Ruby sind diese unter dem Namen Block bekannt. Hier ein Beispiel:

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

Hier übergeben wir einen Block an die Methode *each*. Diese iteriert über alle Elemente der Kollektion, auf der es aufgerufen wird, und führt für jedes Element den übergebenen Block aus. Daher muss der Block auch ein einzelnes Argument akzeptieren. Hier wird deutlich, dass es sich bei Blöcken um anonyme Funktionen handelt.

Anhand von Scala lassen sich einige der anderen Eigenschaften funktionaler Programmierung illustrieren. Das folgende ist eine anonyme Funktion die für einen Integer dessen Nachfolger berechnet (man beachte: der Rückgabetyp wird nicht angegeben, er kann per Type Inference vom Compiler errechnet werden):

Scala
```
x: Int => x + 1
```

Funktionstypen (Funktionssignaturen) lassen sich folgendermaßen angeben:

Scala
```
Int => Int     // ein Int Parameter, Rückgabetyp Int
(Int, Int) => String  // zwei Int Parameter, Rückgabetyp String
```

Dies ist wichtig, wenn man Funktionen höherer Ordnung (Higher Order Functions) definieren möchte, also Funktionen, die andere Funktionen als Parameter akzeptieren. Im Folgenden definieren wir eine Funktion *apply*, die eine Funktion der Signatur *Int => String* erwartet und diese auf ihr zweites Argument *v* anwendet:

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

Ein weiteres Feature, das man aus funktionalen Sprachen kennt, ist Currying. Dabei geht es darum, eine Funktion für einen Teil ihrer Argumente zu evaluieren, was in einer neuen Funktion mit entsprechend weniger Argumenten resultiert. Hier ein Beispiel in Scala.

Scala
```
object CurryTest extends Application {
  def filter(xs: List[Int], p: Int => Boolean): List[Int] = …
  def modN(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  Console.println(filter(nums, modN(2)))
  Console.println(filter(nums, modN(3)))
}
```

filter ist eine Funktion, die aus einer existierenden Liste eine neue Liste erstellt, die allerdings nur die Elemente der ursprünglichen Liste enthält, für die das Prädikat *p true* ist (die Implementierung ist nicht gezeigt). Prädikate sind Funktionen die für ihr einzelnes Argument *true* oder *false* zurückliefern. *modN* ist eine Funktion mit zwei Parametern. Um diese als Prädikat in *filter* verwenden zu können, verwenden wir in den letzen beiden Zeilen Currying: *modN(2)* liefert als Ergebnis eine neue Funktion, bei der *n* an 2 gebunden

ist, und nur noch ein Parameter übrigbleibt. Es entsteht also eine anonyme Funktion, die – wenn man sie ohne Currying geschrieben hätte – so aussehen würde:

```
Scala
mod2(x: Int) = ((x % 2) == 0
```

Dies ist nun ein Prädikat und kann – wie im Beispiel ersichtlich – an *filter* übergeben werden.

Ein ähnliches Beispiel kann man Ruby in auch implementieren, wenn auch nicht ganz so elegant:

```
Ruby
def modN(n)
    return Proc.new {|x| x % n == 0 }
end

mod2 = modN(2)

mod2.call(10)              # returns 0
```

Metaprogrammierung

Ein weiterer Trend in moderneren Sprachen ist Metaprogrammierung [9]. Auch das ist nichts konzeptionell Neues – CLOS [10] hatte bereits vor langer Zeit ein komplettes Metaobjektprotokoll [11], mit dem all die Dinge, die heute gehyped werden, auch schon möglich waren. Nichtsdestotrotz ist Metaprogrammierung eine sehr nützliche Sache. Metaprogrammierung bedeutet, dass ein Programm sich selbst (oder andere Programme) inspizieren und ändern kann.

Auch hier muss man zwischen zwei verschiedenen Konzepten unterscheiden: In statisch getypten, compilierten Programmiersprachen muss der Metaprogrammierungsschritt vor oder während des Compilierungsvorgangs erfolgen: man stelle sich vor, wir fügen zur Laufzeit eine Methode zu einer Klasse hinzu. Diese könnten wir nie aufrufen, weil der Compiler (der ja nichts von der neuen Methode weiß) den Aufruf nicht zulassen würde. In dynamischen Sprachen, insbesondere in denen, die Duck Typing verwenden, kann man natürlich jederzeit eine Methode aufrufen, die man per Metaprogrammierung hinzufügt – sofern das Hinzufügen vor dem Aufrufen erfolgt.

Die statische Metaprogrammierung (auch Compile-Time-Metaprogrammierung genannt) führt ein relatives Nischendasein. Die bekannteste – wenn auch nicht unbedingt schönste – Variante findet sich in C++ mittels der Templatemetaprogrammierung [12]. Sprachen wie Converge [13] oder Template Haskell [14] bieten auch Metaprogrammierungsfeatures.

Dynamische (oder Laufzeit-) Metaprogrammierung ist deutlich weiter verbreitet, insbesondere in Ruby und Groovy. Die Idee ist wie gesagt, dass ein Programm während es läuft selbst sich verändern kann. Hier ist ein Beispiel mit Ruby:

```
Ruby
class SomeClass
```

```
    define_method(“foo”){ puts "foo" }
end
```

define_method definiert eine Methode dynamisch zur Laufzeit. Erwartet werden ein String, der als Methodenname dient, sowie ein Block, der die Implementierung der Methode darstellt. Diese Methode kann dann ganz normal aufgerufen werden:

Ruby
```
SomeClass.new.foo // prints “foo“
```

Es ist auch möglich, den Body der Methode als String zu spezifizieren, und diesen String dynamisch zu evaluieren (mittels *module_eval*).

Sprachen, die Duck Typing unterstützen, bieten auch alle die Möglichkeit, darauf zu reagieren, was passiert, wenn eine Methode oder ein Property angesprochen wird die oder das nicht existiert. Hier ein Beispiel in Groovy:

Groovy
```
class Sammler {
   def data = [:]
   def propertyMissing =
      {String name, value-> data [name] = value }
   def propertyMissing =
      {String name-> data [name] }
}
```

Wir definieren zwei Methoden mit dem Namen *propertyMissing* (die Definition erfolgt durch einen Block!). Die Methode mit einem Argument reagiert auf den Versuch, ein Property zu lesen, die mit zwei Argumenten auf den Schreibzugriff. Die Implementierungen speichern den gesetzten Wert in einem internen Hash. Dies erlaubt den folgenden Client-Code:

Groovy
```
def s = new Sammler()
s.name = "Voelter"
s.vorname = „Markus“
s.name  // is jetzt „Voelter“
```

Es können also beliebige Properties angesprochen werden, als ob sie wirklich in der Klasse definiert wären.

Metaobjektprotokolle erlauben außerdem das Abfangen eines beliebigen Methodenaufrufs; hier ein weiteres Beispiel in Groovy:

Groovy
```
class LoggingClass {
  def invokeMethod(String name, args) {
    println “just executing “+name
    // ausführen der ursprünglichen Methode
  }
}
```

Wenn also eine beliebige Klasse von *LoggingClass* erbt, so werden bei allen Methodenaufrufen vorher zunächst die Methodennamen auf die Konsole ausgegeben. Ja, das sieht aus wie das Standardbeispiel für AOP [16]. Tatsächlich ist die aspektorientierte Programmierung auch aus der Metaprogrammierung hervorgegangen – als Vereinfachung! Heute lesen wir im Netz wieder Artikel, die beschreiben, wie man mit Groovy oder Ruby schön einfach implementieren AOP kann: mittels Metaprogrammierung. Irgendwie drehen wir uns im Kreis…

Domänenspezifische Sprachen

Eine domänenspezifische Sprache (DSL, Domain specific language) [17] ist eine formale, von Programmen (Interpreter, Compiler) verarbeitbare Sprache die dazu dient, einen Teil (oder Aspekt) einer Anwendung zu beschreiben, in einer Notation, die für diesen Aspekt passend ist. Der prominenteste Aspekt im Zusammenhang mit DSLs ist sicher die Business-Logik, es lassen sich aber natürlich auch Architekturabstraktionen, Hardwarezugriffe oder QoS Belange per DSL sinnvoll beschreiben.

Man beachte, dass ein in einer DSL geschriebenes „Programm" zwar formal und präzise sein, jedoch nicht direkt ausführbar sein muss! Ausführbar wird es erst durch einen Interpreter oder einen Codegenerator. Man nennt ein DSL „Programm" daher auch oft Modell oder Spezifikation.

Wichtigstes Charakterisierungsmerkmal bzgl. DSLs ist, ob die DSL bzw. die Modelle in einer Host-Programmiersprache eingebettet ist. Man nennt solche DSLs eingebettete oder interne DSLs. Sind die Modelle eigenständig, redet man von externen DSLs. Ein weiterer wichtiger Aspekt ist, inwiefern vor allem die konkrete Syntax an die Anforderungen frei anpassbar ist. Der Grund eine DSL einzusetzen ist eben oft genau der, dass sich damit Dinge prägnanter ausdrücken lassen. Dazu eigenen sich oft Notationen die in der Domäne sowieso schon gängig sind. Lassen sich diese nicht in die DSL integrieren, ist in vielen Fällen der Nutzen der DSL zweifelhaft.

Interne DSLs sind besonders interessant bei Sprachen die Metaprogrammierung zur „Implementierung" der DSL zulassen, und eine flexible Syntax haben.

Ruby ist in dieser Hinsicht heutzutage sicherlich die am besten geeignete Sprache. Betrachten wir folgendes Beispiel aus Rails, letztendlich eine DSL (oder eine Sammlung von DSLs) zur Implementierung von Webanwendungen: Hier wird eine persistente Datenstruktur namens *Person* als Ruby Klasse deklariert.

Ruby (on Rails)
```
class Person < ActiveRecord::Base
  has_one :name
  has_many :addresses
end

class Address < ActiveRecord::Base
```

```
end
```

Was aussieht wie die Deklaration mit Hilfe eines Schlüsselwortes (*has_one*, *has_many*) ist tatsächlich ein Methodenaufruf: In der Klasse Person wird die Methode *has_one* aufgerufen, sie bekommt einen Parameter – das Symbol *:name* – übergeben. Hier ist eine Alternative Syntax, die den Methodencharakter deutlicher macht:

```
has_one(„name“)
```

Die Tatsache, dass man auch *has_one :name* schreiben kann, verdanken wir Rubys flexibler Syntax, die es erlaubt, Klammern bei Methodenaufrufen unter bestimmten Umständen wegzulassen und bestimmte Strings als Symbol, also mit vorgestelltem Doppelpunkt, zu schreiben.

Was passiert nun aber hier? *has_one* und *has_many* sind statische Methoden, die von der Oberklasse *ActiveRecord::Base* zur Verfügung gestellt werden. Wir rufen sie hier im Rahmen der Klassendefinition von *Person* auf. *has_many* legt Zugriffsmethoden auf eine Liste in der Klasse *Person* an. Der Name der Methoden wird aus dem übergebenen Symbol abgeleitet. Gleiches geschieht mit dem Namensattribut. Wir können dann beispielsweise schreiben

Ruby (on Rails)
```
p = Person.new
p.name = “Markus“
puts p.name # erzeugt die Ausgabe “Markus“
```

Die Implementierung der Zugriffsmethoden ist nicht ganz so einfach, da sie Datenbankzugriffe kapseln (das ist der Sinn des *ActiveRecord*-Frameworks).

Ein weiteres Beispiel für DSLs sind Builder, hier am Beispiel von Groovy erläutert. Auch dies ist eine eingebettete DSL.

Groovy
```
def build = new groovy.xml.MarkupBuilder(writer)
build.html {
  head {
    title 'Hello World'
  }
  body(bgcolor: 'black') {
    h1 'Hello World'
  }
}
```

Was aussieht wie die direkte Baumstruktur eines HTML-Dokuments, ist tatsächlich die clevere Ausnutzung von *methodMissing*, Closures und Hash-Literalen.

Auch statisch getypte Sprachen können sich für interne DSLs eignen. Beispielsweise lassen sich in Scala neue Sprachkonstrukte definieren. Das folgende Beispiel verwendet *loop/unless* wie wenn es ein eingebautes Sprachfeature wäre:

Scala
```
var i = 10;
```

```
loop {
  Console.println("i = " + i)
  i = i 1
} unless (i == 0)
```

Tatsächlich ist es allerdings mittels einer Bibliothek definiert. Dabei kommen die automatische Konstruktion von Closures und die Möglichkeit, Methoden in Operator-Syntax zu verwenden, zum Einsatz. Hier ist der Code (Erklärung unter [18])

Scala

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

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

Eine weitere wichtige Zutat für DSLs ist schlicht und ergreifend ein größerer Zeichenvorrat. Die allermeisten Sprachen verwenden immer noch 7-bit ASCII. Das folgende ist ein gültiges Fortress [19] Programm (entnommen aus [20]).

Fortress

$$
\begin{array}{l}
\mathit{conjGrad}[\![\,\mathrm{Elt}\ \textbf{extends}\ \mathrm{Number},\ \textbf{nat}\ \mathrm{N},\\
\qquad\quad \mathrm{Mat}\ \textbf{extends}\ \mathrm{Matrix}[\![\,\mathrm{Elt}, N\times N\,]\!],\\
\qquad\quad \mathrm{Vec}\ \textbf{extends}\ \mathrm{Vector}[\![\,\mathrm{Elt}, \mathrm{N}\,]\!]\\
\qquad\ ]\!](A:\mathrm{Mat},\ x:\mathrm{Vec}):(\mathrm{Vec},\ \mathrm{Elt})\\
\quad \mathit{cgit}_{\max} = 25\\
\quad z:\mathrm{Vec} = 0\\
\quad r:\mathrm{Vec} = x\\
\quad p:\mathrm{Vec} = r\\
\quad \rho:\mathrm{Elt} = r^{\mathrm{T}} r\\
\quad \textbf{for}\ j \leftarrow \textbf{seq}(1:\mathit{cgit}_{\max})\ \textbf{do}\\
\qquad q = A\,p\\
\qquad \alpha = \dfrac{\rho}{p^{\mathrm{T}} q}\\
\qquad z := z + \alpha\,p\\
\qquad r := r - \alpha\,q\\
\qquad \rho_0 = \rho\\
\qquad \rho := r^{\mathrm{T}} r\\
\qquad \beta = \dfrac{\rho}{\rho_0}\\
\qquad p := r + \beta\,p\\
\quad \textbf{end}\\
\quad (z,\ \|x - A\,z\|)
\end{array}
$$

Auch wenn Sprachen wie Java Unicode unterstützen, so wird in der Praxis doch quasi nur ASCII verwendet. Ein Grund dafür ist, dass man sich schwer tut, Dinge, für die es keine

Taste gibt, einzugeben. Daher besitzt Fortress eine Art Wiki-Syntax um Dinge einzugeben, die sich nicht direkt auf der Tatstatur finden; übrigens ein Vorgehen, das in Tools wie Mathematica [40] schon lange üblich ist.

Die Vorteil interner DSLs sind vor allem die relativ geringe Einstiegshürde und die symbolische Integration mit der Hostsprache. Der zentrale Schwachpunkt ist natürlich die Tatsache, dass die Syntax dadurch begrenzt ist, was die Hostsprache zulässt. Dies ist bei externen DSLs grundlegend anders: Je nach verwendetem Tooling kann die Syntax beliebig gestaltet werden.

Per Definition sind externe DSLs unabhängig von irgendeiner Programmiersprache. Nichtsdestotrotz ist die Erstellung und Verwendung solcher externer DSLs ganz klar ein Trend rund um (Programmier-)Sprachen. Die Erstellung und die Arbeit mit externen DSLs ist auch unter dem Begriff Modellgetriebene Softwareentwicklung (MDSD) bekannt.

Im Zusammenhang mit externen DSLs sind vor allem zwei Aspekte diskussionswürdig: grafische vs. textuelle Syntax sowie Codegenerierung vs. Interpretation.

Die Antwort auf die Frage nach Generierung vs. Interpretation ergibt sich meist aus dem Kontext. Wenn ein System beispielsweise auf einer Legacy-Infrastruktur laufen soll, so muss man in aller Regel Code generieren um den Erwartungen der Infrastruktur gerecht zu werden. Entwickelt man die Plattform gleich mit, kann man die Frage je nach Performance/Codegröße/Deploymentszenario individuell entscheiden. Beschreibt man reines Verhalten (wie bspw. Prozesse, Zustandsautomaten oder Datenvalidierungen) ist Interpretation oft der näherliegende Ansatz. Tools zur Codegenerierung und Modelltransformation sind am Markt vorhanden und ausgereift, z.B. openArchitectureWare [22].

Die Frage nach der konkreten Syntax ist da schon etwas spannender. Bei internen DSLs ist man auf die (hoffentlich flexible) Syntax der Hostsprache beschränkt, was in aller Regel bedeutet, dass es auf textuelle Syntax hinausläuft (wenn man UML Profile als interne DSLs verstehen mag, so ist dies ein Beispiel für eine Hostsprache, die grafische interne DSLs ermöglicht). Bei externen DSLs existiert diese Beschränkung nicht, man hat hier die volle Auswahl: baumartige Konfigurationsmodelle (z.B. Featuremodelle), grafische Notationen (Komponenten, Zustandsmaschinen) oder textuelle DSLs. Mit Tools wie Eclipse GMF [23], MetaEdit+ [24], den Microsoft DSL Tools [25] oder (wenn es denn unbedingt sein muss) UML Profilen lassen sich mit erträglichem Aufwand Editoren für grafische Notationen erstellen. Allerdings haben auch textuelle DSLs ihren Charme: sie lassen sich insbesondere gut in existierende Team-Infrastrukturen integrieren, diff/merge lässt sich bspw. problemlos mit CVS oder SVN erledigen.

Wie sieht es nun bezüglich Tools für textuelle DSLs aus? Natürlich kann man mit antlr [26] oder lpg [27] Grammatiken spezifizieren und Parser generieren. Aber das reicht in der Praxis nicht aus. Man möchte auch für textuelle DSLs die von modernen IDEs bekannten Features nicht missen: Syntax Coloring, Code Completion, Code Folding sowie Echtzeit-

Validierung gehöhren zum Standard. Tools wie INRIA's TCS [38], Eclipse IMP [39] oder oAWs Xtext [22] erlauben genau dies: man spezifiziert die Grammatik seiner Sprache, und ein Generator erstellt einen (im Falle von Xtext Eclipse-basierten) Texteditor, der dem von z.B. Java gewohnten schon recht nahe kommt.

Richtig spannend wird die Arbeit mit DSLs natürlich dann, wenn man grafische und textuelle Notationen beliebig mischen kann, und verschiedene Aspekte des Systems mit verschiedenen, aber miteinander integrierten DSLs beschrieben werden können. Auch merge/diff und Debugger Support sollten verfügbar sein (siehe Martin Fowlers Artikel zum Thema Language Workbenches [28]). Diesem Ziel werden wir mit Tools wie der Intentional Domain Workbench [29] oder JetBrains' Meta Programming System [30] näherkommen. Intentional arbeitet dabei mit einem projizierenden Editor, der beliebige Syntaxformen darstellen kann; insbesondere können Syntaxen und Sprachen nahezu beliebig gemischt werden.

Concurrency

Durch die immer weitere Verbreitung von Multicore Prozessoren wird das Thema Concurreny immer wichtiger und für viele Programmierer unausweichlich. Dummerweise ist nebenläufige Programmierung nicht gerade einfach. Es stellt sich also die Frage, was man tun kann, um dieses Thema leichter handhabbar zu machen. Und natürlich gibt es dafür Lösungsansätze auf Sprachebene.

Pur-Funktionale Sprachen eignen sich hierzu prinzipiell besonders gut, da sie, wie oben schon erwähnt, seiteneffektfrei sind. Dies bedeutet, dass ein Aufruf einer Funktion ausschließlich einen Rückgabewert liefert, es werden keinerlei anderen (globalen) Zustände verändert (Datenstrukturen sind in der Regel immutable, also gar nicht veränderbar: das Hinzufügen eines Elementes zu einer Liste erzeugt eine neue aus der alten Liste und dem Argument). Der Aufrufgraph der Funktionen in einem Programm beschreibt damit sämtliche Abhängigkeiten, es gibt keine versteckten Abhängigkeiten durch veränderbaren (globalen) Zustand. Damit lässt sich ein rein funktionales Programm sehr leicht parallelisieren.

Die aktuellen Mainstream-Sprachen verwenden alle das Shared Memory Modell und Threading. Innerhalb eines Prozesses können (theoretisch) beliebig viele Threads laufen, die sich den Adressraum des Prozesses teilen. Um Race Conditions beim Zugriff auf zwischen Threads gemeinsam genutzte Ressourcen zu verhindern, werden Locks eingesetzt. Entwickler rufen die entpsrechenden Lock-APIs im Code manuell auf. Dies ist natürlich fehleranfällig, weil eine falsche Verwendung von Locks zu Performanceproblemen, Deadlocks und/oder semantisch fehlerhafter Software führt. Um dieses Problem zu entschärfen, zeichnet sich eine Lösung namens Transactional Memory (TM) [31] ab.

Was ist das eigentliche Problem mit der aktuellen Technologie? Der Umgang mit gemeinsam genutzte Ressourcen wird mit einem Protokoll koordiniert – also nicht mit deklarativen Sprachbestandteilen, sondern mit API Aufrufen. Deshalb ist es sehr schwer für Tools (und praktisch unmöglich für den Compiler), Aussagen darüber zu machen, ob der Entwickler das Protokoll korrekt implementiert (was aufgrund der inhärenten nicht-Lokalität in der Praxis sehr schwer sein kann!). Außerdem sind die Protokolle wie wir sie heute verwenden, ein klassischer Fall von Überspezifikation. Statt einfach zu sagen „ich möchte, dass dieser Teil des Systems so ausgeführt würde als ob das System sequentiell liefe“ spezifizieren wir alle möglichen Details im Zusammenhang mit den Protokollen (welches Lock kontrolliert welche Ressource, wann wird es akquiriert, wann wieder freigegeben). Nach diesen Erläuterungen dürfe der Lösungsansatz auf der Hand liegen: Man sollte in der Lage sein, in einem Programm mittels Sprachprimitiven auszudrücken, dass ein bestimmter Block so ausgeführt werden soll, als ob es keine anderen Threads gäbe. Genau das ist – etwas vereinfacht – was Transactional Memory erlaubt:

Fortress
```
atomic do
  // wird ausgeführt wie wenn es
  // nur diesen einen Thread gäbe
end
```

Diese Formulierung hat eben auch noch den weiteren Vorteil, dass man eben nichts über Lock-Details aussagt. Die richtige Allokation von Locks wird damit eine Aufgabe des Compilers und des Laufzeitsystems. Dieses Vorgehen ist ähnlich dem von Garbage Collection (GC). Auch dort überlässt man eine nicht-triviale Aufgabe dem Compiler und Laufzeitsystem – das ist vielleicht nicht in allen Fällen genauso gut wie eine richtige, optimale von-Hand-Implementierung, aber in den allermeisten Fällen ausreichend – und man kann keine Fehler machen (mehr zu der Analogie zwischen TM und GC in [32]).

Übrigens ist Überspezifikation im Zusammenhang mit Nebenläufigkeit generell schlecht. Man denke an folgendes Beispiel:

Java
```
for ( int i=0; i < data.length; i++ ) {
  // do a computation with data[i]
}
```

In dieser Schleife wird eine Iterations-Reihenfolge fest vorgegeben, die möglicherweise gar nicht nötig wäre. Der Compiler kann aber nicht erkennen, ob die Einhaltung der Reihenfolge für die Korrektheit des Programmes relevant ist oder nicht. Man vergleiche das mit folgender Formulierung:

Fortress
```
foreach ( DataStructure ds in data ) {
  // do something with ds
}
```

Hier ist keine Reihenfolge vorgegeben, es wurde nicht überspezifiziert, also kann der Compiler oder das Laufzeitsystem eine Verteilung auf verschiedene Cores vornehmen und

damit die Berechnung parallelisieren. In Fortress (die übrigens auch oben gezeigtes Konstrukt für Transactional Memory enthält) sind Schleifen aus genau diesem Grunde standardmäßig parallel (Beispiel entnommen aus [20]):

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

Falls die sequentielle Verarbeitung wichtig ist, so muss man dies explizit angeben:

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

Fortress geht aber noch einige Schritte weiter: Es lassen sich Maschinenressourcen (Prozessoren, Speicherbereiche) beschreiben und Datenstrukturen sowie Berechnungen explizit oder automatisiert auf diese Ressourcen verteilen. Damit lässt sich effektiv mit mehreren Kernen, Prozessoren oder Maschinen umgehen; natürlich müssen die Compiler und Laufzeitsysteme für Multicore/SMP/Cluster angepasst werden.

Ein radikal anderer Ansatz geht davon aus, dass die Wurzel allen Übels eben gerade in der gemeinsamen Nutzung von veränderbarem Zustand besteht:

- Wenn gemeinsam genutzte Daten nicht modifiziert werden können, muss auch nicht gelockt werden – es kann ja zu keinen inkonsistenten Daten durch gleichzeitige nicht-atomare Veränderung kommen.
- Umgekehrt, wenn keine Daten gemeinsam genutzt werden, muss man auch nichts locken und es gibt dadurch keine Probleme mit Deadlocks etc.

Der erste Punkt – gemeinsame Nutzung nur von nicht-veränderbaren Daten – wird von pur funktionalen Sprachen immer erfüllt, da diese keinen veränderbaren Zustand kennen. Funktionen können außerdem keine Seiteneffekte haben. Damit lassen sie sich leichter parallel ausführen.

Den anderen Punkt erreicht man durch Abschaffung von gemeinsam genutztem Speicher. Die Kommunikation zwischen parallel laufenden Entitäten passiert ausschließlich über den Austausch von Nachrichten. Dies ist bekannt als Actor-Modell [33]. Es gibt verschiedene Implementierungen dieses Paradigmas in verschiedenen Sprachen, die derzeit bekannteste ist sicherlich Erlang [34].

Erlang ist eine funktionale Sprache, die bereits vor ca. 20 Jahren bei Ericsson entwickelt wurde. Sie ist optimiert für verteilte, ausfallsichere und echtzeitfähige (Telekom-) Systeme. Im Kern steht eben wie gesagt ein Actor-basiertes Modell für Nebenläufigkeit, wobei die

Aktoren hier Process heißen (und nichts mit Betriebssystemprozessen oder Threads zu tun haben!).

Mittels *spawn* lässt sich in Erlang ein neuer Prozess starten. Als Argument bekommt *spawn* eine Lamda-Expression übergeben. Der Rückgabewert von *spawn* ist die Prozess-ID (Pid) des neuen Prozesses.

Erlang
```
Pid = spawn(fun() -> doSomething() end)
```

Um einem Prozess eine Nachricht zukommen zu lassen, verwendet man die Ausrufezeichen-Notation. Nachrichten sind dabei nichts Besonderes, sondern jede beliebige Erlang-Datenstruktur – in aller Regel Tupel.

Erlang
```
Pid ! Message
```

Auf Empfängerseite bietet die Sprache so etwas wie ein Unix-select [35] an. Mittels Pattern-Matching [36] wird das ankommende Tupel untersucht; wird ein passender Match gefunden, werden die Elemente des Tupels an die freien Variablen im Match-Ausdruck gebunden (Variablen beginnen mit einem großen Buchstaben, Konstanten mit einem kleinen). Danach werden die Expressions rechts des Pfeils ausgeführt, wobei die nun gebundenen Variablen verwendet werden können.

Erlang
```
loop
  receive
    {add, Id, Name, FirstName} -> ActionsToAddInformation;
    {remove,Id} -> ActionsToRemoveItAgain;
    ...
    after Time -> TimeOutActions
end
```

Um in einer Programmiersprache ein effizientes Actor-Modell zu implementieren werden folgende Zutaten benötigt: Closures um den aktuellen (ggfs. unterbrochenen) Zustand der Berechnung zu speichern sowie (effizientes) Pattern Matching um zwischen den verschiedenen ankommenden Nachrichten unterscheiden zu können.

Interessant ist nun, dass Scala genau diese beiden Features besitzt. Zusammen mit der Möglichkeit, den ! Operator zu überladen, lassen sich in Scala Programme schreiben, die nicht nur fast genauso aussehen wie die Originale in Erlang, sondern auch ähnlich effizient laufen [37]:

Scala
```
def loop: unit = {
  receive {
    case Add(name, firstName) => …
    case Remove(name, firstName) =>…
    case _ => loop(value)
  }
}
```

Plattformen

Bei der Diskussion über Sprachen und deren Features sollte man im Auge behalten, dass die Frage nach Laufzeitumgebungen davon (mehr oder weniger) unabhängig ist.

Beispielsweise kommt in diesem Artikel sicherlich eine gewisse Skepsis bezüglich der Weiterentwicklung von Java als Programmiersprache zum Ausdruck. Ich denke tatsächlich, dass Java seinen Zenit überschritten hat und die Sprachinnovation im Kontext anderer Sprachen passiert. Dies gilt aber nicht für die Java Virtual Machine (JVM)! Die JVM ist inzwischen eine extrem stabile und hochoptimierte Ausführungsplattform die mit Sicherheit eine große und lange Zukunft vor sich hat (gleiches gilt im Prinzip auch für die .NET Common Language Runtime (CLR)). Die JVM muss sich dabei nur sehr begrenzt weiterentwickeln (man denke an die Diskussion im *invokedynamic* [41] oder die Unterstützung für Tail Recursion [42]).

Gleiches gilt für Enterprise Plattformen wie bspw. Java Enterprise Edition (JEE). JEE bietet quasi ein Betriebssystem für Enterprise Anwendungen, mit Fokus auf Skalierbarkeit, Deployment, standardisierte Betriebsverfahren usw. Es ist unwahrscheinlich, dass die Investitionen in diese Infrasktrukturen „einfach so" weggeworfen werden. Meiner Meinung nach ist es ein zentraler Vorteil von Groovy/Grails gegenüber Ruby/Rails, dass sich erstere mit den etablierten Plattformen integriert.

Tooling

Auch wenn ich in diesem Artikel vor allem auf Sprachfeatures und –konzepte eingehen will, so kann man das Thema Tooling nicht völlig aus den Augen lassen. Tooling umfasst nicht nur Editoren (inkl. Syntax Coloring, Code Completion und Refactoring) sondern auch die Möglichkeit, Programme statisch zu analysieren und Fehlermeldungen auszugeben.

Es ist allgemein bekannt, dass sich für statisch typisierte Sprachen leichter komfortable Editoren bauen lassen, weil das Typsystem viele Informationen liefert die der Editor auswerten kann. Allerdings gibt es inzwischen auch erfolgreiche Bemühungen, den IDE Support für dynamische Sprachen zu verbessern, wie beispielsweise Netbeans' Unterstützung für Ruby zeigt. Es wird aber immer Grenzen geben. Vor allem automatisiertes Refactoring ist auf ein aussagekräftiges, statisches Typsystem angewiesen.

Eine interessante Betrachtung ist der Zusammenhang zwischen Toolunterstützung und Metaprogrammierung. Es ist kein Zufall, dass die allermeisten Metaprogrammierungssysteme dynamischer Natur sind, wo IDEs traditionell keine so wichtige Rolle gespielt haben. Wenn sich das Programm zur Laufzeit selbst verändert, dann kann eine IDE natürlich nichts davon wissen und keine Unterstützung bieten. Systeme mit statischer (also compile-time) Metaprogrammierung bieten keinen wirklichen IDE Support.

Gleiches gilt auch für interne DSLs. In Converge, wo man beliebigen DSL Code in das Programm einbetten kann (der dann während der Compilierung in einen regulären Converge-AST überführt wird) gibt es keinerlei Editor Support für derartige DSLs. Natürlich könnte man sich so etwas prinzipiell vorstellen – dann wäre aber der Aufwand zur Spezifikation einer solchen DSL erheblich höher. In Ruby – einem Vertreter der Sprachen, die interne DSLs unterstützen – gibt es keinen IDE Support für die DSLs.

Meiner Meinung nach stellt das den zentralen Schwachpunkt interner DSLs dar: Die IDE der Hostsprache bietet keinerlei Unterstützung beim Schreiben des „DSL Programms". Dabei geht es nicht nur um bunten Code oder Ctrl-Leertaste. Insbesondere können auch keine domänenspezifischen Fehlermeldungen ausgegeben werden sondern nur Fehler auf der Abstraktionsebene der Hostsprache. Da eine DSL ja dafür gedacht ist von Domänenexperten verwendet zu werden, ist dies ein ernsthaftes Problem.

Bei externen DSLs ist dies natürlich völlig anders. Der IDE Support ist oft sehr gut, da man ja sowieso spezifische Editoren für die DSL erstellen muss. Dort ist es auch problemlos möglich, domänenspezifische Fehlermeldungen (sprich: Constraintvalidierung) einzubauen. Tools wir GMF (für grafische DSLs) oder Xtext (für textuelle DSLs) machen das Erstellen von DSL-spezifischen Editoren mit vertretbarem Aufwand möglich.

Schlussendlich möchte ich noch auf kurz das Thema Analysierbarkeit eingehen. Tools zur statischen Codeanalyse sind ja derzeit auch mächtig im kommen. Besonders interessant wird das Thema im Zusammenhang mit Concurrency. Je mehr man Dinge die man traditionell mittels Protokollen implementiert hat in Sprachfeatures überführt (bzw. beim Library-Design explizit auf Analysierbarkeit achtet) desto mehr kann man mittels Analysewerkzeugen erreichen. Ein sehr schönes Beispiel für diesen Ansatz findet sich im Rahmen des Singularity Forschungsprojekts bei Microsoft [46]. Details bitte selbst nachlesen in [47]. Lohnt sich!

Zusammenfassung

Der übergreifende Trend ist, Sprachen zu definieren, die möglichst wenige, aber dafür mächtige Konzepte besitzen. Diese mächtigen Konzepte dienen dann dazu mit den Sprachmitteln Bibliotheken zu definieren, deren Verwendung sich so anfühlt, als ob es Spracherweiterungen wären. Statt Aktoren wie bei Erlang als Teil der Sprache zu definieren, werden sie in Scala in einer Bibliothek definiert. Metaprogrammierung und flexible Syntax in Ruby sind eine weitere Möglichkeit, dieses Ziel zu erreichen.

Der Spruch „man soll jedes Jahr eine neue Sprache lernen" war immer wahr, und ist es heutzutage umso mehr, da derzeit ziemlich viel Innovation im Sprachbereich (außerhalb der Forschung) stattfindet. Man muss ja nicht gleich ein Experte in allen möglichen Sprachen sein, aber es lohnt sich, den Blick über den Tellerrand zu wagen um neue Ideen und Blickwinkel zu bekommen.

In den Referenzen habe ich eine ganze Reihe interessanter Links zusammengefasst. Ich möchte auch kurz auf den Software Engineering Radio Podcast hinweisen, dort gibt es zu vielen der hier erwähnten Konzepte ausführliche Episoden (www.se-radio.net).

Danke…

an Nora Ludewig, Bernd Kolb, Arno Haase, Sven Efftinge, Achim Demelt, Michael Wiedeking, und Martin Lippert für das Feedback zu diesem Artikel

Referenzen

[0] http://www.gotw.ca/publications/concurrency-ddj.htm
[1] http://en.wikipedia.org/wiki/Type_system
[2] http://www.scala-lang.org/
[3] http://msdn2.microsoft.com/en-us/vcsharp/aa336745.aspx
[4] http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx
[5] http://www.ruby-lang.org/
[6] http://groovy.codehaus.org/
[7] http://en.wikipedia.org/wiki/Type_inference
[8] http://en.wikipedia.org/wiki/Functional_programming
[9] http://en.wikipedia.org/wiki/Metaprogramming
[10] http://en.wikipedia.org/wiki/Common_Lisp_Object_System
[11] http://www.lisp.org/mop/index.html
[12] http://en.wikipedia.org/wiki/Template_metaprogramming
[13] http://convergepl.org/
[14] http://www.haskell.org/th/
[15] http://www.pluralsight.com/blogs/dbox/archive/2006/05/12/23354.aspx
[16] http://en.wikipedia.org/wiki/Aspect-oriented_programming
[17] http://en.wikipedia.org/wiki/Domain-specific_programming_language
[18] http://www.scala-lang.org/intro/targettyping.html
[19] http://research.sun.com/projects/plrg/
[20] http://research.sun.com/projects/plrg/PLDITutorialSlides9Jun2006.pdf
[21] http://www.wolfram.com/
[22] http://www.openArchitectureware.org
[23] http://www.eclipse.org/gmf/
[24] http://www.metacase.com/
[25] http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx
[26] http://www.antlr.org/
[27] http://sourceforge.net/projects/lpg/
[28] http://martinfowler.com/articles/languageWorkbench.html
[29] http://www.intentsoft.com
[30] http://www.jetbrains.com/mps/
[31] http://en.wikipedia.org/wiki/Software_transactional_memory
[32] http://www.cs.washington.edu/homes/djg/papers/analogy_oopsla07.pdf

[33] http://en.wikipedia.org/wiki/Actor_model
[34] http://www.erlang.org/
[35] http://en.wikipedia.org/wiki/Select_(Unix)
[36] http://en.wikipedia.org/wiki/Pattern_matching
[37] http://lampwww.epfl.ch/~odersky/papers/jmlc06.pdf
[38] http://wiki.eclipse.org/index.php/TCS
[39] http://www.eclipse.org/proposals/imp/
[40] http://www.wolfram.com/products/mathematica/index.html
[41] http://jcp.org/en/jsr/detail?id=292
[42] http://en.wikipedia.org/wiki/Tail_recursion
[43] http://en.wikipedia.org/wiki/Duck_typing
[44] http://www.inquiry.com/techtips/cpp_pro/10min/10min0100.asp
[45] http://www.devsource.com/article2/0,1759,1544618,00.asp
[46] http://research.microsoft.com/os/singularity/
[47] http://pages.cs.wisc.edu/~remzi/Classes/838/Fall2001/Papers/singularity-tr05.pdf