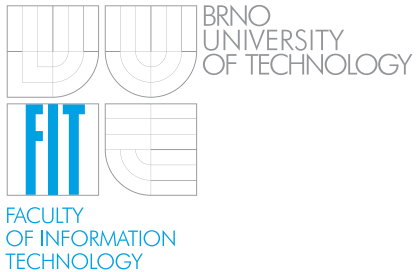
IVS : Programovací Jazyky a Paradigmata

Svatopluk Šperka

Ústav počítačové grafiky a multimédií
Fakulta informačních technologií

Vysoké učení technické v Brně



Jazyk jako nástroj

- Jazyk, který používáme, ovlivňuje co a jak můžeme vyjádřit.
- Možná i to, co můžeme myslet – Sapir-Whorfova hypotéza.
- Existuje široké spektrum programovacích jazyků postavených na různých perspektivách, vlastnostech, podporovaných konstrukcích a v neposlední řadě s různým způsoby zápisu.
- Povaha řešeného problému by měla ovlivnit výběr jazyka – nástroje – k jeho řešení.

Genealogie Programovacích jazyků

http://www.levenez.com/lang/lang.pdf

Člověk vs. Počítač

- Způsob, jakým se programuje počítač na úrovni HW je komplikovaný proces s velmi primitivní základní jednotkou - instrukcí.
- Pro člověka je tento pohled nepřirozený, protože neprimitivní jevy není možné snadno pojmout na takto nízké úrovni rozlišení.
- Jazyky lze umístit do spektra podle toho, jak moc reflektují povahu fungování počítače a na druhou stranu přírozenost či úroveň myšlení člověka.

Efektivita

- Efektivita *vývoje*.
- Efektivita *výsledku* (čas, paměť).
- Pro každý program je dobré zvážit, co je důležitější.
- Adekvátní jazyk/prostředí může mnohonásobně zvýšit produktivitu.
- Výsledek může být pomalejší, ale stále plně přijatelný.
- Navíc může být srozumitelnější, přehlednější a udržovatelnější.

Imperativní vs. Deklarativní

- **Imperativní**
  - stav programu a sekvence kroků, které tento stav mění a tím provádějí výpočet
  - program specifikuje, jak se problém řeší
- **Deklarativní**
  - popis, specifikace problému danými konstrukty
  - vyhodnocovací mechanismus najde řešení.
  - program specifikuje, co se má řešit
  - funkcionální a logické paradigma
  - *SQL*
- Řada jazyků nabízí oba přístupy.

- **Statické**
  - kompilací je dáno vše, co se může v programu objevit a dít.
  - obtížné zkoumat a měnit stav programu
  - nelze přidávat nové elementy (funkce, objekty, typy) či měnit ty stávající
  - optimalizace v době kompilace
  - cyklus edit-compile-run-debug
- **Dynamické**
  - co se bude dít se rozhoduje za běhu
  - program je možno jednoduše zkoumat, rozšiřovat a manipulovat.
  - optimalizace v době běhu
  - je možno program vyvíjet interaktivním způsobem

- **Manuální**
  - je nutno ručně pamět alokovat a uvolňovat
  - komplikované v komplikovaných programech
  - memory leak je velice častá chyba v programech
- **Automatická**
  - programátor paměť pouze alokuje
  - mechanismus alokace sleduje alokovanou paměť
  - její nepoužívané části v definované momenty uvolňuje *garbage collector*

Paradigmata

- Procedurální
- Funkcionální
- Logické
- Objektově orientované
- Konkurentní
- ...

Imperativní

- To co nejspíš znáte - např. *C*, *Pascal*.
- Explicitní stav programu měněn sekvencí příkazů.
- Reflektují charakteristiky architektury moderních počítačů.
- Dále je rozvíjeno v procedurálním, strukturovaném či modulárním paradigmatu (týká se organizace kódu).
- Konstrukce jako cykly, větvení, ukazatele, záznamy, *procedury* (*funkce*).
- Možnosti optimalizací - dosažení výkonných implementaci algoritmů.

Funkcionální

- Inspirací je lambda kalkulus (Alonso Church, 1941).
- Fundamentální konstrukt je funkce v matematickém slova smyslu (občané první kategorie).
- Žádná explicitní manipulace dat. Data proplouvají funkcemi a jsou tak transformována.
- *Specifikace problému* v podobě funkcí.
- Žádné vedlejší efekty (kromě I/O operací).
- Vyhodnocování v libovolném pořadí (narozdíl od imperativních jazyků).
- Snažší ukázat korektnost programů.
- Adekvátní úroveň abstrakce pro mnoho problémů.
- Prototypování algoritmů a datových struktur.

Funkcionální jazyky

- LISP
  - Dynamicky typovaný.
  - Hodně závorek.
  - *Scheme*, *Clojure* (běží na JVM)
- Haskell
  - Silně typovaný.
  - Líné vyhodnocování (jen když je potřeba).
  - Vysoce expresivní.
- *F#* (.NET), *Scala* (JVM)

Test prvočíselnosti v Clojure

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))
```

Binarní Strom v Haskellu

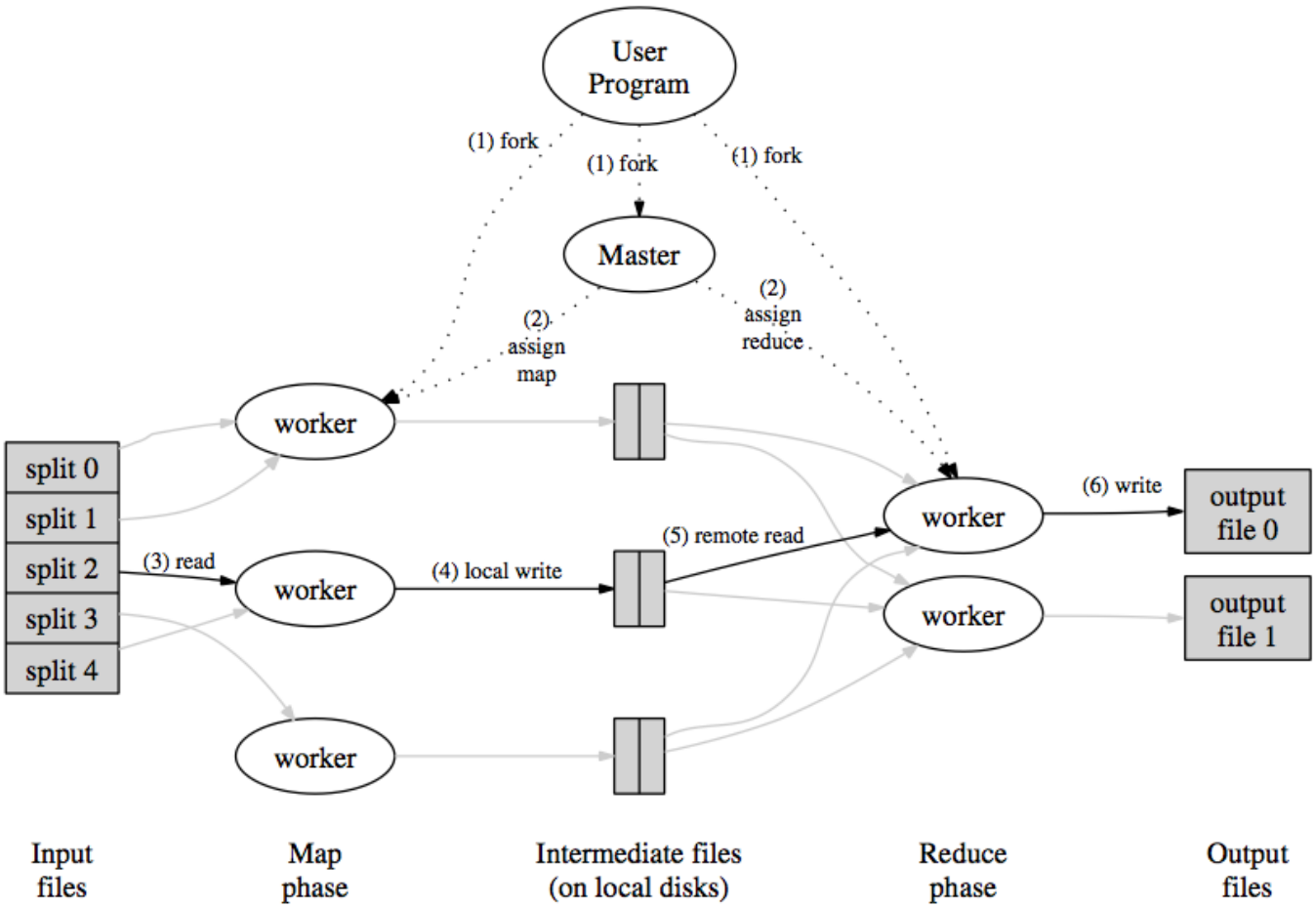
```haskell
data Tree a = Tip | Node a (Tree a) (Tree a)
        deriving (Show,Eq)

leaf x = Node x Tip Tip

t2 = Node 17
        (Node 12 (leaf 8) (leaf 15))
        (Node 46 Tip (leaf 57)

add value Tip = leaf value
add value t@(Node x xl xr)
        | value < x = Node x (add value xl) xr
        | value > x = Node x xl (add value xr)
        | otherwise = t

size Tip = 0
size (Node _ tl tr) = 1 + size tl + size tr

inOrder Tip = []
inOrder (Node x xl xr) = inOrder xl ++ x : inOrder xr
```

- Metoda paralelizace výpočtu, která fundamentálně vychází z funkcionálního programování.
- funkce *map* – `map(in_key, in_value) -> list(out_key, intermediate_value)`
- funkce *reduce* – `reduce(out_key, list(intermediate_value)) -> list(out_value)`
- Vstupní dvojice jsou namapovány na mezi-dvojice.
- Dvojice, jež jsou výstupem *map* jsou zhluknuty podle hodnoty klíče. Tyto jsou následně vstupem funkce *reduce*.

Počítaní slov pomocí MapReduce

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Schéma MapReduce



[4]

FLP – Funkcionální a Logické Programování

(v magisterském studiu)

Haskell, Prolog

- Inspirace přírodou (buňky).
- **Objekty** a **zprávy** jako základní modelovací konstrukty.
- Objekty zapouzdřují svůj stav a komunikují pomocí zpráv.
- Objekt umožňuje stavět vyšší celky přičemž skrývá přebytečnou komplexitu, která není nutná pro jeho použití.
- Dva fundamentální přístupy k organizaci objektů a znovupoužití kódu – *prototypový* a *třídní*.

... Smalltalk is NOT only its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.

The big idea is "messaging" ...

Alan Kay [1]

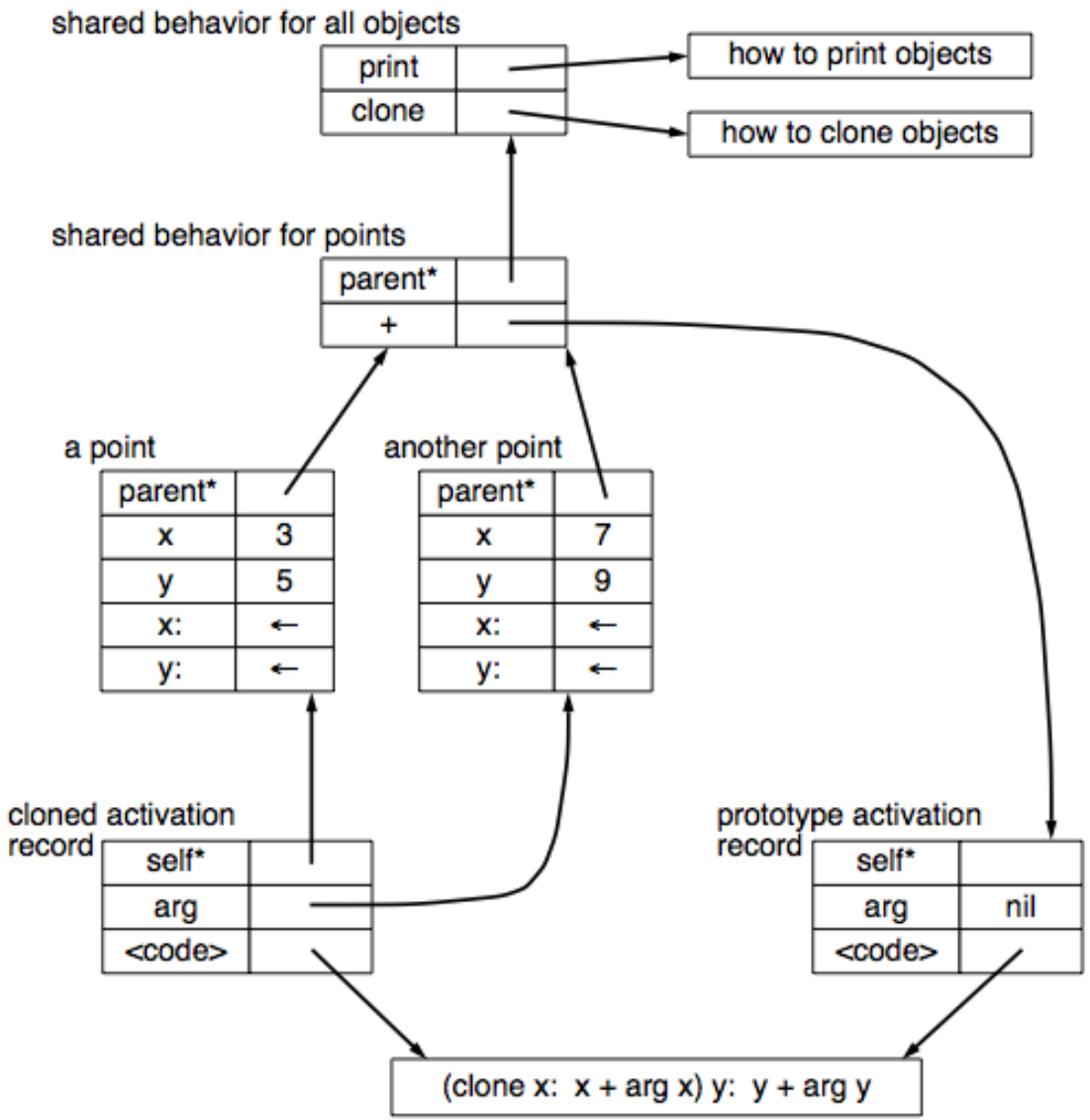
[1] http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

Třídní Přístup

- Třídy jsou šablony, předobrazy, platónské ideje pro své instance.
- Třídy organizovány v hierarchii dedičnosti.
- Specializace/generalizace.
- *Smalltalk*, *Scala*, *C#*, *Java*, *C++*, ...

Prototypy

- Každý objekt je v principu jedinečný.
- Objekty sdílejí určité rysy (traits).
- Delegace nahrazuje dedičnost.
- Objekty je možné kopírovat a vytvářet množiny objektů stejné struktury.
- Podporuje průzkumný styl programování pomocí experimentování a prototypování (zdola-nahoru).
- *Self*, *JavaScript*


shared behavior for all objects
print
clone
how to print objects
how to clone objects
shared behavior for points
parent*
+
a point
parent*
x
3
y
5
x:
←
y:
←
another point
parent*
x
7
y
9
x:
←
y:
←
cloned activation record
self*
arg
<code>
prototype activation record
self*
arg
nil
<code>
(clone x: x + arg x) y: y + arg y

Prototypy a delegace v JS

```
var point_trait = {
        plus : function(arg) {return new point(this.x + arg.x, this.y + arg.y)}
}

function point(x, y) {
        this.x = x; this.y = y
}
point.prototype = point_trait;

var finite_plane_point_trait = {
        dim_x : 5,
        dim_y : 5,
        plus : function(arg) {
                tx = this.x + arg.x; ty = this.y + arg.y;
                return new point( tx <= this.dim_x ? tx : this.dim_x,
                        ty <= this.dim_y ? ty : this.dim_y)
        }
}

var a = new point(1, 2);
var b = new point(2, 4);

a.plus(b)
// (3, 6)

// Firefox, Chrome
a.__proto__ = finite_plane_point_trait

a.plus(b)
// (3, 5)
```

DJA - Dynamické Jazyky
(v magisterském studiu)
Lisp, Smalltalk, Self

IIS - Informační Systémy
JavaScript

ICP – Seminář C++
IJA – Seminář Java

Skriptovací jazyky

- Propojování komponent a nástrojů.
- Automatizace úloh.
- Nabízejí interaktivní prostředí a rychlý vývoj.
- Vysokoúrovňové, expresivní jazyky.
- Kombinují více paradigmat (imperativní, objektové, funkcionální).
- Dynamické a interpretované (často bytekód).
- Pokročilé možnosti práce s textem.
- *Perl*, *Python*, *Ruby*, *PHP*, *Lua* ..

Počítaní slov v Ruby

```ruby
counts = Hash.new( 0 ) # výchozí počet pro každé slovo

File.open( filename, "r" ) { |f|
  f.each_line{ |line|
    # odstranění speciálních symbolů
    line.gsub!( /([.,:()'])/," " )

    line.split.each{ |word|
      counts[word] += 1
    }
  }
}

counts.each_pair{ |word, count| # key, value
  puts "#{word} : #{count}"
}
```

ISJ – Skriptovací jazyky

Perl, Python, Ruby

- Vícejádrové procesory a distribuované systémy.
- Potřeba škálovatelnosti a robustnosti.
- Konvenčně se používá *sdílená paměť* a zajištění výlučného přístupu.
- Alternativní metoda *zasílání zpráv*.
- Způsob jak stavět masivní distribuované systémy.

Aktorový Model

- model konkurentního výpočtu (Hewitt, 1973)
- vše je aktor
- v reakci na zprávu může aktor
  - poslat konečný počet zpráv jiným aktorům
  - vytvořit konečný počet nových aktorů
  - změnit své chování, kterým bude reagovat na další zprávy

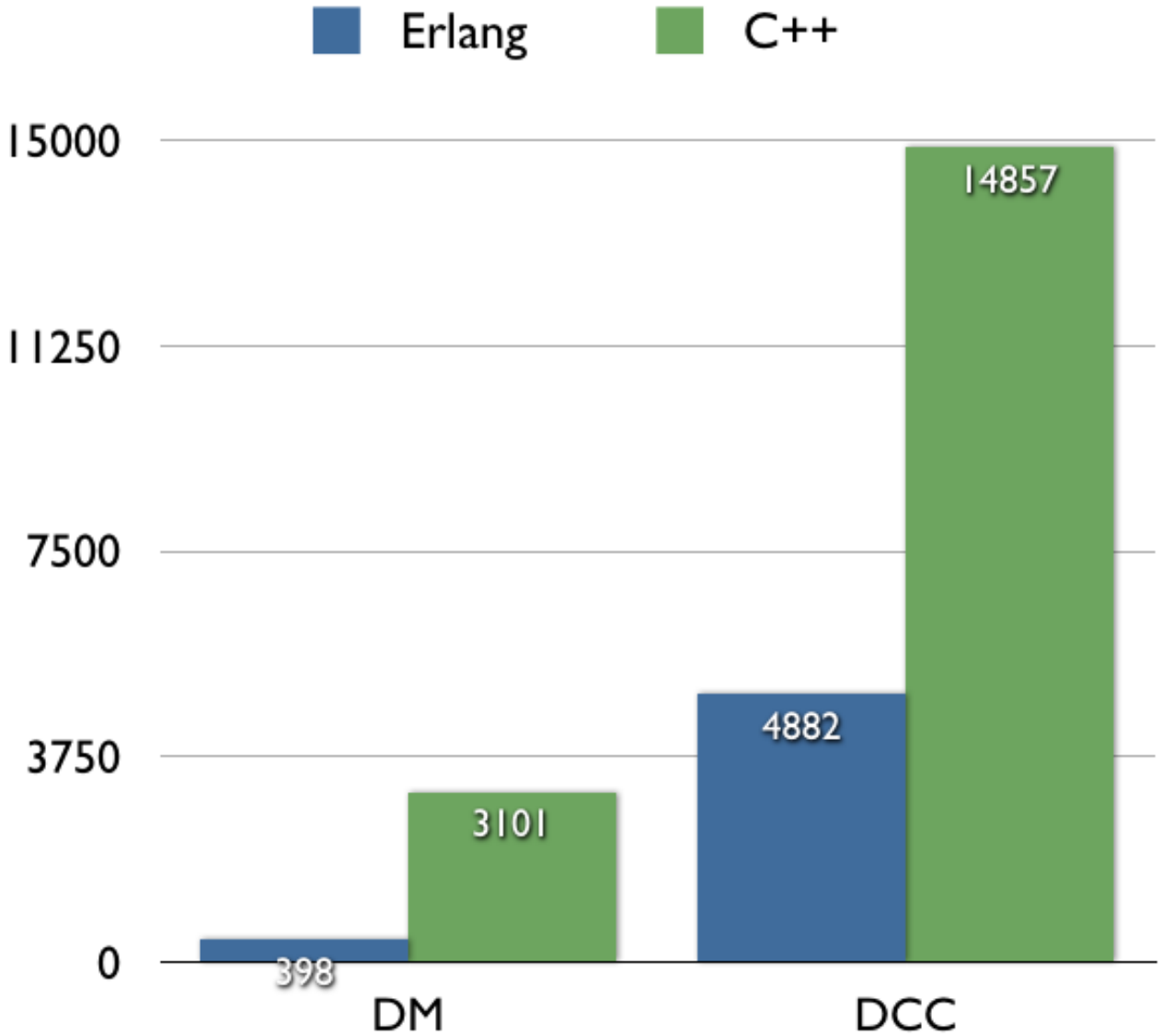
Erlang

- Navržen pro vývoj robustních, škálovatelných soft-realtime systému (Ericsson, telekomunikace).
- Sekvenční podmnožina je funkcionální.
- Aktorový model jako model konkurentního/distribuovaného výpočtu (procesy).
- Komunikace jen pomocí zpráv, *žádná* sdílená pamět.
- Změny v kódu za běhu.
- Není rozdíl v komunikace s procesy na stejném a na jiném uzlu.
- Filosofie “let-it-fail” (nech to spadnout a vyrovnej se s následky).

Řetězec procesů v Erlangu

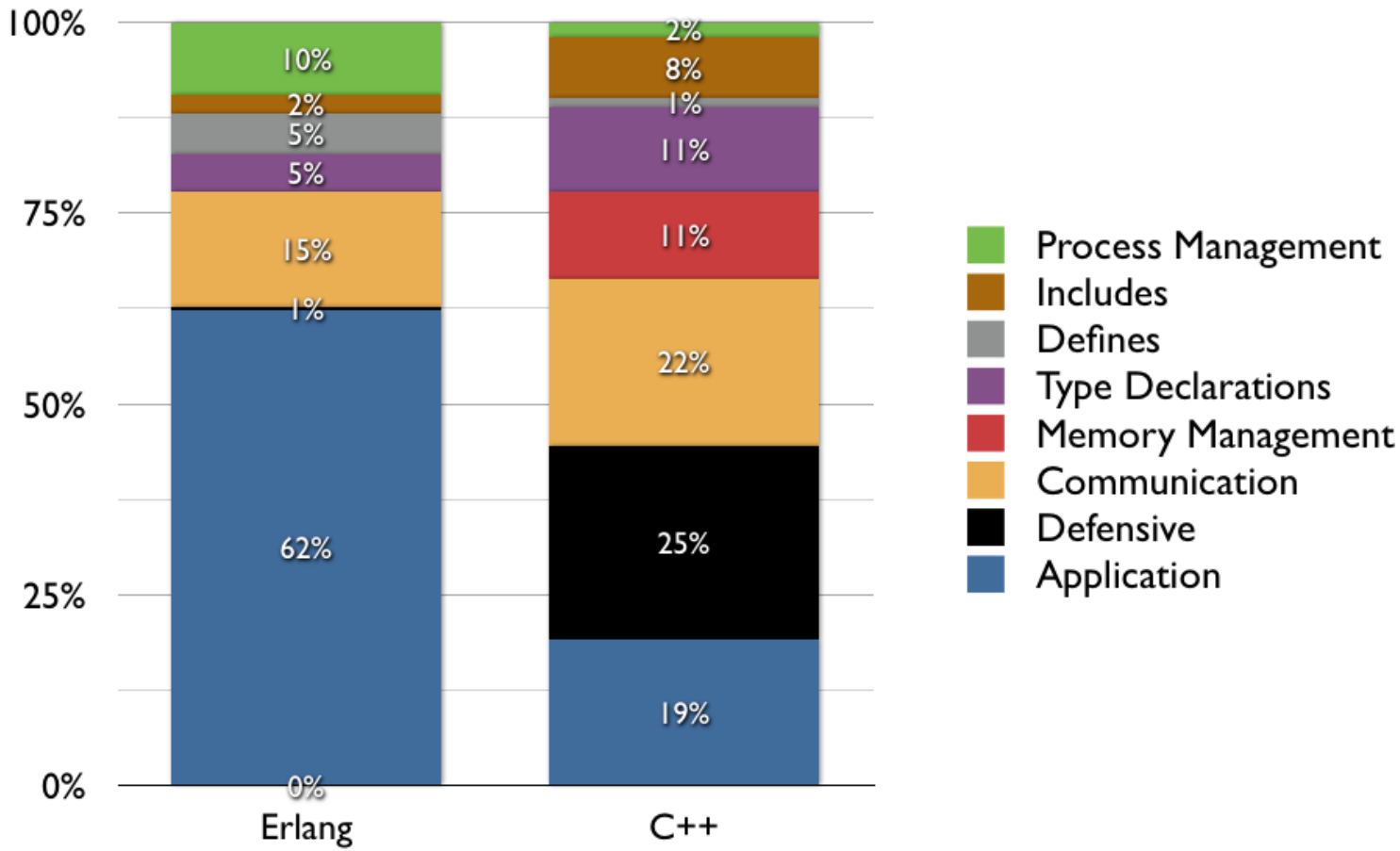
```
triggerChain( ProcessCount ) ->
    PID = spawn( ?MODULE, element, [ProcessCount, self()] ),
    PID ! ping,
    receive
        ping ->
            PID ! ping,
            receive ping -> okend
    end.

element( 1, First ) ->
    receive ping ->
        First ! ping,
        receive ping -> First ! ping end
    end;
element( Remaining, First ) ->
    receive
        ping ->
            PID = spawn( ?MODULE, element, [Remaining - 1, First]),
            PID ! ping,
            receive ping -> PID ! ping end
    end.
```

Počet řádku kódu



Zdroj: [3]

Proporce kódu



Výtky vůči Javě

- Nutnost neustále uvádět typy
- Málo syntaktického cukru
- Není čistě objektový
- Nejsou uzávěry či lamda abstrakce (anonymní funkce)
- Ztrácí kontakt s konkurencí jako je např. C# pro .NET

Alternativy

- Groovy
- Scala
- Ceylon

Groovy

- Dynamický jazyk, dynamické typování
- Kompiluje do bytecodu JVM
- Plně interoperabilní (dokonce zaměnitelný)
- Uzávěry, konstrukce na uzavíraní externích prostředků
- Metaobjektový protokol, reflexe
- Vhodný pro doménově specifické jazyky (DSL) či psaní skriptů
- Relativně pomalý

```
// práce se souborem

myFileDirectory = "C:\\temp\\"
myFileName = "myfile.txt"
myFile = new File(myFileDirectory + myFileName)

printFileLine = { println "File line: " + it }

myFile.eachLine( printFileLine )


// dynamické přidání vlastnosti

Number.metaClass.getProperty = { propertyName ->
    if (propertyName == "percent") { delegate / 100 }
    else { delegate }
}

assert 20.5.percent == 0.205
assert 20.percent == 0.2
```

Scala

- *Sca*lable *La*nguage – vhodný pro malé i velké programy
- Plně objektově orientovaný
- Funkce občany první kategorie a podporuje funkcionální styl programování
- Staticky typovaná s odvozováním typů
- Úspornější než Java
- Podpora pattern matchingu, rysů (traits)
- Komplexní a expresivní jazyk

Java vs. Scala

```scala
// třída ve Scale

class Sample(x: Int, val p: Int) {
  def instMeth(y: Int) = x + y
}

object Sample {
  def staticMeth(x: Int, y: Int) =
    x * y
}
```

```java
// třída v Javě

class Sample {
  private final int x;
  public final int p;

  Sample(int x, int p) {
    this.x = x; this.p = p;
  }

  int instMeth(int y) {
    return x + y;
  }

  static int staticMeth(int x, int y)
    return x * y;
  }
}
```

```scala
// pattern matching
def simplifyTop(expr: Expr): Expr = expr match {
        case UnOp("-", UnOp("-", e)) => e // Double negation
        case BinOp("+", e, Number(0)) => e // Adding zero
        case BinOp("*", e, Number(1)) => e // Multiplying by one
        case _ => expr
}
```

Literatura

- [1] A. Tucker, R. Noonan: *Programming Languages*. McGraw-Hill. 2002.
- [2] C. Bisciglia, A. Kimball & S. Michels-Slettvet: *MapReduce Theory and Implementation.* 2007. http://code.google.com/edu/submissions/mapreduce/llm3-mapreduce.ppt
- [3] J. H. Nyström et al.: *High-level Distribution for the Rapid Production of Robust Telecoms Software: Comparing C++ and Erlang.* Concurrency and Computation 20. 2008.
- [4] Dean, J. and Ghemawat, S.: *MapReduce: Simplified data processing on large clusters*. 2008.
- [5] D. Ungar: *SELF: The Power of Simplicity*. 1991. http://labs.oracle.com/self/papers/self-power.html
- [6] F. Cesarini, S. Thompson: *Erlang Programming, O'Reilly Media*, 2009
- [7] M. Odersky et al.: *Programming in Scala (2nd ed.)*, Artima Press, 2010
- [8] H. Abelson and G. J. Sussman: Structure and Interpretation of Computer Programs http://mitpress.mit.edu/sicp/full-text/book/book.html

Odkazy

- Steve Yegge: *Dynamic Languages Strike Back*. http://steve-yegge.blogspot.com/2008/05/dynamic-languages-strike-back.html; http://www.youtube.com/watch?v=tz-Bb-D6teE
- *Programming is Hard, Let's Go Scripting...* http://www.perl.com/pub/2007/12/06/soto-11.html?page=1
- *The Language List* http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm
- *Real World Haskell* http://book.realworldhaskell.org/read/
- *What is Erlang-Style Concurrency?* http://ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/
- *Erlang @ Facebook* http://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf
- *Revenge of the Nerds* http://www.paulgraham.com/icad.html
- Gavin King: *Ceylon* http://relation.to/Bloggers/Ceylon
- Structure and Interpretation of Computer Programs in Clojure. http://sicpinclojure.com/