

Programming Paradigms

Unit 15 — Concurrent Programming with Erlang

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

- 1 Serial vs. Parallel Programs
- 2 Basic Concepts of Erlang
- 3 Lists and Tuples

Outline

- 1 **Serial vs. Parallel Programs**
- 2 Basic Concepts of Erlang
- 3 Lists and Tuples

Serial vs. Parallel/1

- The programming world used to be much simpler
 - Almost all programs ran in a **serial** fashion on a **single CPU**
 - A programmer would not have to deal with parallelism or concurrency
- Developers could rely on the fact that more powerful CPUs would appear, making their code run faster

Serial vs. Parallel/2

- However, things have changed
 - At some point we will reach the limits of Moore's law
 - Designers of CPUs are facing severe problems with power dissipation (due to high integration)
- One solution to these problems is to build multi-core and/or distributed systems
- In order to fully exploit these systems, programmers have to embrace a **different programming style**

Multi-threaded Programming

- In many programming languages, such as Java or C++, **threads** were introduced to parallelize programs
- This should lead to better performance, as multiple cores can actually be utilized
- However, there is a downside to multi-threading
 - Threads **share resources**
 - Resource **contention** leads to bottlenecks
 - Writing (and debugging) multi-threaded code is very **complex**

Concurrent Programming in Erlang

- We are going to look at **Erlang**, a programming language specifically designed for **concurrency**
- Erlang is a **compiled concurrent functional** programming language (with some roots in Prolog)
- The name has two interpretations:
 - **Ericsson Language**; it was originally developed at Ericsson
 - **Agner Karup Erlang** was a Danish mathematician, whose work was used in telephone network analysis
- In 1986, Joe Armstrong developed the first version at Ericsson



Concurrency

- Put simply, **concurrent** means that tasks can be done in any order without compromising the results
 - For example, shuffling two decks of cards: can be done in **any order** or even in parallel
- This is not the same as parallelism:
 - concurrent tasks do not have to be done in parallel;
 - however, we can **run them in parallel without any negative effects.**

Performance

- Performance was not one of the main features of the languages we looked at so far
- Erlang was developed with **high performance** in mind
- It has to satisfy the requirements of telecom companies:
 - **Hundreds of thousands of processes** have to run in a highly distributed environment
 - Processes should not be able to **corrupt each other's memory**
 - Systems cannot be taken down to upgrade software, i.e., **hot-swapping of code** has to be possible
 - Has to be able to deal with **crashing processes reliably**

Erlang in Industry

- Erlang has been progressed in the 1990s to become an industry-strength language that today is used in many different projects
 - CouchDB, a document-based database
 - SimpleDB, a distributed database that is part of Amazon Web Services
 - Mnesia, a distributed database
 - Zotonic, a content management system and web framework
 - Chat service in Facebook, handling more than 200 million active users
 - Messaging servers in WhatsApp, achieving up to 2 million connected users per serve
 - T-Mobile uses Erlang in its SMS and authentication systems
 - Goldman Sachs, high-frequency trading programs

Outline

- 1 Serial vs. Parallel Programs
- 2 Basic Concepts of Erlang**
- 3 Lists and Tuples

Processes

- The basic building block of Erlang is a **process**
 - Can be seen as an **agent running a piece of code**
 - This is done **concurrently** to other processes, every process running at its own pace
- Processes **don't share** any resources
 - Frequently also referred to as **lightweight** processes
- Erlang processes/programs run on a **virtual machine** that **automatically adapts** to the underlying hardware
 - Runs on multiple cores, multiple CPUs in a distributed system, or on a single CPU

Process Communication

- Sometimes processes have to interact with each other
- This is done via **message passing**, i.e., copying some information and sending it to another process
 - This happens even if both processes are running on the same machine
- This makes it possible to distribute Erlang applications quite easily
- Has the downside that things that are simple to formulate in other languages are a bit harder in Erlang
- As a programmer you have to start thinking in terms of **concurrent processes**

Fault Tolerance

- Rather than trying to achieve perfect error handling, Erlang follows the philosophy of “Let it crash”
- Part of the system goes down in a controlled way, and then is restarted from a clean state
- The event leading to a crash and the crash are logged, so this can be analyzed to find a possible fault

Functional Programming

- Erlang borrows a couple of concepts from **functional programming**, though it is not a “pure” functional language
- In particular, “**functional programming**” in Erlang means
 - Programs are built entirely out of **functions**, no objects anywhere
 - Functions will (usually) return the **same values**, given the same inputs
 - Functions will (usually) not have **side effects**, i.e., they don't modify program state
 - A variable can only be assigned a value **once**

Getting Started in Erlang

- Although most programs will be compiled, we'll start out with the **interactive Erlang shell**
- Start the Erlang interpreter at the command line by typing `erl`, like this:

```
carbon$ erl
Erlang R16B03 (erts-5.10.4) ...

Eshell V5.10.4 (abort with ^G)
1>
```

- Now you can execute basic statements

Hello World in Erlang

- Here's the obligatory “Hello World!” program in Erlang

```
> io:format("Hello World!\n").  
Hello World!  
ok
```

- Each statement ends with a **period** ('.') at the end!

Standard Data Types and Operators

- Erlang knows the **standard data types** and **operators**

```
> 2 + 2.
```

```
4
```

```
> 2 + 2.0.
```

```
4.0
```

```
> "This is a string".
```

```
"This is a string"
```

- Erlang has **strong typing**

```
> 4 + "string".
```

```
** exception error:
```

```
bad argument in an arithmetic expression
```

Atoms

- Erlang has some similarities with Prolog
- An **atom** is a literal (constant with a name) and is used to refer to real-world things
- An atom starts with a lower-case letter, or need to be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains a space

```
> hello.
```

```
hello
```

```
> phone_number.
```

```
phone_number
```

```
> 'Monday'.
```

```
'Monday'
```

```
> 'phone number'.
```

```
'phone number'
```

Variables and Pattern Matching/1

- Here are some more similarities to Prolog
- **Variables** start with an upper-case letter
- **Pattern matching** is used to bind variable to values using the = operator
- The = operator is the **match operator** (and not an assignment operator)
 - The left-hand side **pattern** is matched against a right-hand side **term**
 - If successful, any unbound variables in the pattern become bound

```
> X.
```

```
** 1: variable 'X' is unbound **
```

```
> X = 2.
```

```
2
```

```
> X + 1.
```

```
3
```

- Variables can only be **instantiated once** (similar to Haskell)

```
> X = 1.
```

```
** exception error: no match of right hand side value 1
```

- Pattern matching can also be used with more complex structures (see later)

Variables and Pattern Matching/2

- Notice that the pattern with the **unbound variables** must be on the **left-hand side** of the match operator
 - i.e., the expression on the right-hand side must be fully instantiated

```
> X = 1.
```

```
1
```

```
> 1 = Y
```

```
variable 'Y' is unbound
```

- Variables cannot be matched to variables (unlike in Prolog)

```
> X = Y.
```

```
variable 'Y' is unbound
```

- The right-hand term can also be an expression that is first evaluated

```
> X = 1 + 2.
```

```
3
```

Outline

- 1 Serial vs. Parallel Programs
- 2 Basic Concepts of Erlang
- 3 Lists and Tuples**

Lists/1

- Erlang knows **lists**, which similar to Prolog are enclosed in brackets

```
> [1,2,3].
```

```
[1,2,3]
```

```
> [72,97,32,72,97,32,72,97].
```

```
"Ha Ha Ha"
```

```
> [1,2,72,97,32,72,97,32,72,97].
```

```
[1,2,72,97,32,72,97,32,72,97].
```

```
> [9,12].
```

```
"\t\f"
```

- Printable characters (e.g., ≥ 32 and ≤ 126) are shown (if all elements are printable)

Lists/2

- A list can contain any number of elements, with possibly **different types**

```
> [1,2,3,a,b,c,"hello"].  
[1,2,3,a,b,c,"hello"]
```

- Lists can be **nested**, containing other lists

```
> [this, list, contains, a, [list]].  
[this, list, contains, a, [list]]  
  
> [1, [3, [5]]].  
[1, [3, [5]]]
```

- The function **length()** returns the length of a list

```
> length([this, list, contains, a, [list]]).  
5  
  
> length([]).  
0
```


Concatenating and Subtracting Lists

- The list **concatenation operator** `++` appends its second argument to its first and returns the resulting list.

```
> [1,2,3]++[a,b,c].  
[1,2,3,a,b,c]
```

- The list **subtraction operator** `--` produces the difference of two lists as follows:
 - for each element in the second argument, the first occurrence of this element (if any) is removed from the first argument.

```
> [1,2,3,2,1,2]--[2,1,2].  
[3,1,2]
```

```
> [2,1,2]--[1,2,3,2,1,2].  
[]
```

Lists and Pattern Matching

- Pattern matching can be applied to lists with **head** and **tail** elements
 - Like in Prolog, the "|" operator separates the head from the tail

```
> [Head|Tail] = [1,2,3,4].
```

```
[1,2,3,4]
```

```
> Head.
```

```
1
```

```
> Tail.
```

```
[2,3,4]
```

```
> [A,B,C] = [1,2,3]
```

```
[1,2,3]
```

```
> A.
```

```
1
```

- Differently to Prolog, Erlang will not do an exhaustive search to match all possible values
 - There is no backtracking to bind variables to other values such as in Prolog

List Comprehension/1

- Similar to Haskell, Erlang supports **list comprehension**

```
> [X || X <- [1,2,a,3,4,b,5,6], X > 3].  
[a,4,b,5,6]
```

- i.e., the list of X such that X is taken from the list [1,2,a,...] and X is greater than 3
- The expression X <- [1,2,a,...] is a **generator**
- The expression X > 3 is a **filter**

List Comprehension/2

- An additional filter, `integer(X)`, can be added to restrict the result to integers

```
> [X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].  
[4,5,6]
```

- Generators can be combined, e.g., the Cartesian product of two lists

```
> [{X, Y} || X <- [1,2,3], Y <- [a,b]].  
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

Tuples/1

- **Tuples** are fixed-length (heterogeneous) lists and enclosed in curly braces

```
> Origin = {0, 0, "null point"}.  
{0, 0, "null point"}.
```

- Tuples can be **nested**

```
> Person = {person, {name,"Joe Smith"}, {age,24}}.  
{person, {name,"Joe Smith"}, {age,24}}
```

- As shown in this example, tuples are often used as **maps** or **hashes**
 - Atoms are used for the hash keys and strings (or other data types) for the values

Tuples/2

- The function `element/2` extracts individual elements from a tuple

```
> element(1,Person)
person
> element(2,Person)
{name,"Joe Smith"}
```

- The function `setelement/3` sets an element in a tuple

```
> P2 = setelement(3,Person,{age,25}).
{person, {name,"Joe Smith"}, {age,25}}
> P2.
{person, {name,"Joe Smith"}, {age,25}}
> Person.
{person, {name,"Joe Smith"}, {age,24}}
```

- Notice that the variable `Person` has not been modified, only the new variable `P2`

Tuples/3

- The function `tuple_size` returns the size of a tuple

```
> tuple_size(Person)
```

```
3
```

```
> tuple_size({})
```

```
0
```

Tuples and Pattern Matching

- Pattern matching can be used to extract values from a tuple

```
> Person = {person, {name,"Joe Smith"}, {age,24}}.  
{person, {name,"Joe Smith"}, {age,24}}
```

```
> {person, {name,Name}, {age,Age}} = Person  
{person, {name,"Joe Smith"}, {age,24}}
```

```
> Name.  
"Joe Smith"
```

```
> Age.  
24
```

- In the first line, variable `Person` has been bound to a tuple
- In the second line, the variables `Name` and `Age` are bound to a string and a number, respectively