Maps and Structs

Johan Montelius

KTH

VT23

Java vs Elixir

```
public static int fib(int n) {

    if ( n == 0 )
      return 0;

    int n2 = 0, n1 = 1;
    int ni = n1;

    for(int i = 1; i < n; i++) {
        ni = n1 + n2;
        n2 = n1;
        n1 = ni;
     }
     return ni;
};
```

```
def fib(0) do  0 end
def fib(1) do  1 end
def fib(n) do
  fib(n-1) + fib(n-2)
end
```


Elixir type specification

```
@spec fib(integer()) :: integer()

def fib(0) do  0 end
def fib(1) do  1 end
def fib(n) do
  fib(n-1) + fib(n-2)
end
```

The compiler does not care about type specifications!

Compiles ok:

```
      :
   fib(:bananas)
      :
```


types in Elixir

What types do we have:

Singletons, the types of individual data structures:

- 1, 2 or 42
- :foo, :bar or :atom
- {:foo, 42}

Unions of singletons, what we normally refer to as "types":

- integer(): any integer value
- float(): any floating point value
- atom(): any atom
- pid(): a process identifier
- reference(): a reference
- fun(): a function
- .. and many more

Could also be written without the "()".

types in Elixir

Types for compound data structures:

- tuples: `{}`, `{atom(), integer()}`, ...
- lists: `[integer()]`, `[{atom(), integer()}]`, `[]`...
- tuple(): a tuple of any size
- list(): a proper list of any type (`[any()]`
- list(integer()) : a proper list of integers

type declarations

Cards are represented as `{:card, suit, value}`, where the suit is represented using the atoms `:spade`, `:heart`, `:diamond` and `:clubs`.

How do we specify the type for *suit/1*:

```
suit({:card, suit, _}) do suit end
```

```
@spec suit(tuple()) :: atom()
```


defining types

We would like to define our own type that specifies what a card looks like.

```
@type value() :: 1..13
```

```
@type suit() :: :spade | :heart | :diamond | :clubs
```

```
@type card() :: {:card, suit(), value()}
```

```
@spec suit(card()) :: suit()
```


defining types

```
@type boolean() :: true | false
```

```
@type byte() :: 1..255
```

```
@type number() :: integer() | float()
```

The type `any()`, defines the union of all types.

defining types

The type list(t) is the type of lists containing elements of type t.

```
@type list(t) :: [] | [t|list(t)]
```

```
@type string() :: list(char())
```

Define the type of a deck of cards.

```
@type deck() :: list(card())
```


program annotation

Type specifiers are used for:

- documentation of intended usage
- automatic detection of type errors

the compiler does not check types

Dialyzer:

- checks that given specifications agree with call patterns
- detects exceptions and dead code

dynamically typed

Elixir is a *dynamically typed* language: types are checked and handled at run time.
other dynamically typed languages: PHP, Python, Erlang, Lisp, Prolog

Java is a statically typed language: types are checked and handled at compile time.
other statically typed languages: C/C++, Haskell, Scala, Rust

statically typed

The advantage of a statically typed language:

```
typedef struct person {
  int id;
  char name[20];
  char email[20];
} person;

void hello(person *who) {
  printf("Hello %s\n", who->name);
}
```

```
@type person() :: {:person,
                   integer(),
                   binary(),
                   binary()}

def hello({:person, _, name, _}) do
   IO.write("Hello #{name}\n")
end
```

In a statically typed language, the compiled code of `hello()` *takes the structure* `person` *for granted.*

type inference

A statically typed language does not imply that the programmer has to specify all types explicitly - the compiler can infer the types (Haskell, Rust, ..).

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```


dynamically vs statically

The pros and cons of dynamically typed languages:

- pro: quick to write code
- pro: compiling an easier task
- con: induces an overhead at run-time
- con: errors detected first at run-time (and maybe very late)

So why is Elixir dynamically typed?
Easier to handle dynamic code updates in distributed systems.

Type systems

- Elixir is a dynamically typed programming language.
- External tools (Dialyzer) can check for type errors.
- Type specification, if correct, helps in understanding the code.
- Dynamically vs statically typed systems - pros and cons.

problem

```
 {:car, "Volvo",
        {:model, "XC60", 2018},
        {:engine, "A4", 4, 2000, 140},
        {:perf, 4.6, 8.8}}
```

```
def car_brand_model( {:car, brand, {:model, model, _}, _ , _}) do
  "#{brand} #{model}"
end
```

key-value list

```
 {:car, "Volvo",
        [{:model, "XC60"},{:year 2018}, {:engine, "A4"},
         {:cyl, 4}, {:vol, 2000}, {:power 140},
         {:fuel, 4.6}, {:acc 8.8}]}
```

```
def car_brand_model( {:car, brand, prop} ) do
    case List.keyfind(prop, :model, 0) do
      nil ->
         brand
      {:model, model} ->
         "#{brand} #{model}"
    end
end
```


key-value list

What is the asymptotic time complexity of `keyfind/3`*?*

alternative syntax: `[model:  "XC60", year:  2018, ...]`

introducing Maps

An efficient implementation of a key-value store with a syntax for pattern matching.

- `%{}` : an empty map
- `myCar = %{:brand => "Volvo", :model => "XC60", :year= 2008}` : define properties
- `%{:model => model} = myCar` : pattern matching
- `newCar = %{myCar | :year => 2018}` : map as template for new map

Still no compiler support to detect errors.

introducing Structs

```
defmodule Car do

  defstruct brand: "",  year: 0,  model: "",  cyl: 0,   power: 0

  def brand_model(%Car{brand: brand, model: model}) do
    "#{brand} #{model}"
  end

  def year(car = %Car{}) do
    car.year
  end

end
```

Requesting a property that is not defined is detected at compile time.

Summary

- dynamically and statically typed systems: pros and cons
- tuples: simple but gives us some problems
- key-value lists: what problems do we solve, what remains
- Maps: pattern matching and more efficient
- Structs: towards the advantage of a statically typed system