Erlang Introduction 2.

László Lövei

Department of Programing Languages and Compilers
Faculty of Informatics
Eötvös Loránd University

May 21, 2009

Outline

Funs

Funs are function objects which may be used as any other data.

Fun expression

```
lists:filter(fun (N) when N rem 2 == 0 -> true;
                 (_) -> false
             end,
             [1,2,3,4,5,6])
```

- The example function returns true for even numbers
- There may be any number of arguments
- There may be any number of clauses (at least one)
- Named functions may be referred too: `fun add/2`

Calling funs

Using funs

```
filter(F, []) -> [];
filter(F, [Hd|Tl]) ->
    case F(Hd) of
        true -> [Hd | filter(F, Tl)];
        _    -> filter(F, Tl)
    end.
```

- Call syntax for funs is the same as for named functions

Calling funs

Using funs

```
filter(F, []) -> [];
filter(F, [Hd|Tl]) ->
    case F(Hd) of
        true -> [Hd | filter(F, Tl)];
        _    -> filter(F, Tl)
    end.
```

- Call syntax for funs is the same as for named functions
- Guards for funs:
  - `is_function(F)`
  - `is_function(F, Arity)`

Exercises

1. Implement the `map` function! It has two arguments: `F`, a fun, and `L`, a list. It should return a list that consist of the results of calling `F` on the elements of `L`.
2. Generalize the `sum` function! It should get a new argument, which specifies the operation to be used instead of addition.

Tail recursion

Tail call example

```
filter(F, [Hd|Tl]) ->
    case F(Hd) of
        true -> [Hd | filter(F, Tl)];
        _    -> filter(F, Tl)
    end.
```

- A tail call is a function call in the last position of a function

Tail recursion

Tail call example

```
filter(F, [Hd|Tl]) ->
    case F(Hd) of
        true -> [Hd | filter(F, Tl)];
        _    -> filter(F, Tl)
    end.
```

- A tail call is a function call in the last position of a function
- Other calls grow the runtime stack, because the caller function must be continued

Tail recursion

Tail call example

```
filter(F, [Hd|Tl]) ->
    case F(Hd) of
        true -> [Hd | filter(F, Tl)];
        _    -> filter(F, Tl)
    end.
```

- A tail call is a function call in the last position of a function
- Other calls grow the runtime stack, because the caller function must be continued
- Tail calls are optimized in Erlang: they do not use stack space

Tail recursion

Tail call example

```
filter(F, [Hd|Tl]) ->
    case F(Hd) of
        true -> [Hd | filter(F, Tl)];
        _    -> filter(F, Tl)
    end.
```

- A tail call is a function call in the last position of a function
- Other calls grow the runtime stack, because the caller function must be continued
- Tail calls are optimized in Erlang: they do not use stack space
- Recursive tail calls are important in long-running server code

Tail recursion

Factorial function

```
fact(0) -> 1;
fact(N) when N>0 -> N*fact(N-1).
```

- Not tail recursive: the result has to be processed

Tail recursion

Factorial function

```
fact(N) when N>=0 -> fact(N, 1).
fact(0, F) -> F;
fact(N, F) -> fact(N-1, N*F).
```

- The usual solution is the introduction of an accumulator

Exercises

1. Create a tail recursive variant of the `map` function!
2. Create an Erlang function that
   - reads lines from the keyboard (see `io:get_line`),
   - prints the number of words for every line, and
   - stops when an empty line is entered.

   Make sure the function is tail recursive!

Record motivation

Using structured data

```
new(Name, Age, Phone) -> Name, Age, Phone.
is_adult({_Name, Age, _Phone}) ->
    Age >= 18.
new_phone({Name, Age, _Phone}, NewPhone) ->
    {Name, Age, NewPhone}.
```

- Tuples can be used to store structured data, but they are clumsy
- Easy to make mistakes
- Hard to extend the structure
- Large tuples are very inconvenient

Record usage

Using structured data

```
-record(person, name, age, phone).
new(Name, Age, Phone) ->
    #person{name=Name, age=Age, phone=Phone}.
is_adult(#person{age=Age}) -> Age >= 18.
new_phone(P=#person{}, NewPhone) ->
    P#person{phone=NewPhone}.
```

- The record definition contains the record name and field names

Record usage

Using structured data

```
-record(person, name, age, phone).
new(Name, Age, Phone) ->
    #person{name=Name, age=Age, phone=Phone}.
is_adult(#person{age=Age}) -> Age >= 18.
new_phone(P=#person{}, NewPhone) ->
    P#person{phone=NewPhone}.
```

- The record definition contains the record name and field names
- Constructors take the field values

Record usage

Using structured data

```
-record(person, name, age, phone).
new(Name, Age, Phone) ->
    #person{name=Name, age=Age, phone=Phone}.
is_adult(#person{age=Age}) -> Age >= 18.
new_phone(P=#person{}, NewPhone) ->
    P#person{phone=NewPhone}.
```

- The record definition contains the record name and field names
- Constructors take the field values
- Fields are usually accessed by pattern matching

Record usage

Using structured data

```
-record(person, name, age, phone).
new(Name, Age, Phone) ->
    #person{name=Name, age=Age, phone=Phone}.
is_adult(#person{age=Age}) -> Age >= 18.
new_phone(P=#person{}, NewPhone) ->
    P#person{phone=NewPhone}.
```

- The record definition contains the record name and field names
- Constructors take the field values
- Fields are usually accessed by pattern matching
- Updating fields has its own syntax

Record usage

Using structured data

```
-record(person, name, age, phone).
new(Name, Age, Phone) ->
    #person{name=Name, age=Age, phone=Phone}.
is_adult(#person{age=Age}) -> Age >= 18.
new_phone(P=#person{}, NewPhone) ->
    P#person{phone=NewPhone}.
```

- The record definition contains the record name and field names
- Constructors take the field values
- Fields are usually accessed by pattern matching
- Updating fields has its own syntax
- Records are turned into tagged tuples at compile time

Exercises

1. Define a record representation for complex numbers!
2. Create functions for complex number operations like addition, conjugation, absolute value!

List comprehensions

Squares of even numbers

```
[A*A || A <- lists:seq(1, 10), A rem 2 == 0]
```

- Generators match a pattern on every element of a list

List comprehensions

Squares of even numbers

```
[A*A || A <- lists:seq(1, 10), A rem 2 == 0]
```

- Generators match a pattern on every element of a list
- Filters evaluate conditions

List comprehensions

Squares of even numbers

```
[A*A || A <- lists:seq(1, 10), A rem 2 == 0]
```

- Generators match a pattern on every element of a list
- Filters evaluate conditions
- When the patterns match and the conditions are true, an expression is evaluated

List comprehensions

Squares of even numbers

```
[A*A || A <- lists:seq(1, 10), A rem 2 == 0]
```

- Generators match a pattern on every element of a list
- Filters evaluate conditions
- When the patterns match and the conditions are true, an expression is evaluated
- The result if the list of the evaluation results

Exercises

Write a list comprehension that

1. calculates every Pythagorean triple below a given limit!
2. converts the upper case letters to lower case in a string!

Binaries

- Binary data is an uninterpreted sequence of bytes
- Binary constructor syntax: `<<1,2,3>>`
- Character data may be specified: `<<"ABC">>` yields `<<65,66,67>>`
- Field size can be specified in bits: `<<1:32>>` yields `<<0,0,0,1>>`
- Field type can be specified: `<<0.5/float>>` yields `<<63,224,0,0,0,0,0,0>>`
- Embedded binaries may be used to concatenate them: `<<A/binary, B/binary>>`

Binary patterns

Sum of 32 bit signed integers

```
sum32(<<First:32/signed, Tail/binary>>) ->
    First + sum32(Tail);
sum32(<< >>) -> 0;
sum32(_)          -> throw(bad_align).
```

- Read the first 32 bits

Binary patterns

Sum of 32 bit signed integers

```
sum32(<<First:32/signed, Tail/binary>>) ->
    First + sum32(Tail);
sum32(<< >>) -> 0;
sum32(_)         -> throw(bad_align).
```

- Read the first 32 bits
- Continue with the rest of the data

Binary patterns

Sum of 32 bit signed integers

```
sum32(<<First:32/signed, Tail/binary>>) ->
    First + sum32(Tail);
sum32(<< >>) -> 0;
sum32(_)          -> throw(bad_align).
```

- Read the first 32 bits
- Continue with the rest of the data
- Stop when there is no more data

Binary patterns

Sum of 32 bit signed integers

```
sum32(<<First:32/signed, Tail/binary>>) ->
    First + sum32(Tail);
sum32(<< >>) -> 0;
sum32(_)          -> throw(bad_align).
```

- Read the first 32 bits
- Continue with the rest of the data
- Stop when there is no more data
- Signal an error if the last data chunk is not 32 bit long

Exercise

Create a function that reads the contents of a file into a binary (see `file:read_file`), and counts the lines in the text!