Refactoring Erlang Programs[1]

Zoltán Horváth, László Lövei, Zoltán Csörnyei,
Tamás Kozsik, Anikó Víg, Tamás Nagy, Roland Király,
Melinda Tóth, Róbert Kitlei

Dept. Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary

Central-European Functional Programming School (CEFP 2007)

[1] Supported by ELTE IKKK (GVOP-3.2.2.-2004-07-0005/3.0) and Ericsson Hungary.

Contents

Erlang

- Target area: telecommunication software
  - Concurrent and distributed
  - Fault tolerant
  - Soft real-time
- Other key features:
  - Functional (high-level declarative syntax)
  - Dynamic type system
  - Light-weight processes
  - Extensive support libraries (OTP)

Examples

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

gcd(A,0) -> A;
gcd(A,B) -> gcd(B, A rem B).

gcd(A,B) -> if B==0 -> A;
              true -> gcd(B, A rem B)
            end.
```

Data in Erlang: Terms

- *Number*: integers and floats

```
0      42      -9
3.14        2.3e-4
```

- *Atom*: string-like data used as literals
  - alphanumeric, begins with lowercase letter
- *Variable*: terms can be bound to variables

```
Pi = 3.14          Label = yellow
```

  - Variable names begin with an uppercase letter
  - Variables are not variable!

Compound terms

- *List*
- *Tuple*
- *Record*
- *String*

Tuple

- Fixed-sized, immutable array of terms
- Enclosed in braces

```
Tup = {date, 2007, 6, 22}
```

```
size(Tup) == 4
element(2, Tup) == 2007
```

List

- Variable-sized collection of terms
- Enclosed in brackets
- Head plus tail

```
Short = [1,2]
Long = [ 0 | Short ]
```

```
Long == [0, 1, 2]
length(Short) == 2
hd(Short) == 1
tl(Short) == [2]
Short ++ [3] == [1, 2, 3]
```

String

- List of ASCII codes
- Usual special syntax

```
"Bar" == [$B, $a, $r] == [66, 97, 114]
```

Record

- Relatively new feature
- Similar to tuple
- Fields are given names

```
-record(date,[year,month,day]).

DateOfBirth = #date{month=4,day=28,year=1970}
NextBirthday = DateOfBirth#date{year=2008}
```

Control flow

- Sequence: evaluated for side effects

```
distance( {X1,Y1},  {X2,Y2} ) ->
   DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

- Branching on patterns:

```
length(L) -> case L of
                 [] -> 0;
                 [Hd|Tl] -> 1 + length(Tl)
             end.
```

- Branching on conditions:

```
if X > Y -> greater;
   X < Y -> less
end
```

Recursion / loop

- Recursive function definitions

```
length(L) -> case L of
                 [] -> 0;
                 [Hd|Tl] -> 1+length(Tl)
             end.
```

- Loops by tail-recursion

Binding variables

- Each variable can be bound only once in its lifetime
- Multiple binding occurrences are possible

```
gcd(X,X) -> X;
gcd(X,Y) ->
    if X > Y -> {NewX,NewY} = {X-Y,Y};
       true  -> {NewX,NewY} = {X,Y-X}
    end,
    gcd(NewX,NewY).
```

Functions

- Name is an atom, arguments are patterns
- Result is computed by clauses
- The first clause with a matching pattern is executed

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

```
sum([]) ->
    0;
sum([Hd | Tl]) ->
    Hd + sum(Tl).
```

Pattern matching

- Formal arguments of function clauses

```
sum([]) -> 0;
sum([Hd | Tl]) -> Hd + sum(Tl).
```

- Branching expressions

```
length(L) -> case L of
                 [] -> 0;
                 [Hd|Tl] -> 1+length(Tl)
             end.
```

- Binding variables

```
Long = [ 0, 1, 2 ],    [0, Short] = Long
```

Pattern matching

- Multiple roles
  - Decomposes terms
  - Looks up values
  - Binds variables
- Same syntax as a term, but may contain unbound variables
- Puts an implicit condition on the matched term
  - Match failures generate runtime errors

Modules

- Functions are compiled in modules
- Module name: atom (same as the file name)
- Export list: functions callable from outside

```
factorial:fact(5) == 120
```

```
-module(factorial).
-export([fact/1]).
fact(1) -> 0;
fact(N) -> N * fact(N-1).
```

Types

- No types declared for functions and variables
- Function signature: module, name, arity
- Dynamic typing
- Pattern matching might result in run-time error
- Flexibility, polymorphism
- Branching on patterns:

```
case get_value() of
    {X, Y} -> tuple;
    [X, Y] -> list
end
```

Standard environment

- Built-in functions (BIFs): implemented in the runtime system
  `length`, `element`, `is_tuple`, `atom_to_list` etc.

- STDLIB: a set of useful modules

```
math:sqrt(25) == 5.0000
lists:seq(3,6) == [3,4,5,6]
string:substr("foobar", 2, 3) == "oob"
io:format("~b: ~s~n", [42, answer])
```

Lambda expressions

- Unnamed functions stored/passed as data

```
F = fun(X) -> X+1 end
F(3) == 4
```

- Useful for writing or using generic functions

```
lists:map(F, [1,2,3])    == [2,3,4]

lists:filter( fun(X) -> X > 0 end
            , [2, -3, 4, -1]
            )
== [2,4]
```

Concurrency

- Based on light-weight processes
- Communicate by message passing
- The spawn BIF starts a new process

```
Pid = spawn(fun() -> main() end)
```

- A process can be assigned a name (an atom)

```
register(johnny_b_good, Pid)
```

- Distribution: processes on different Erlang nodes can communicate transparently

Communication

- Processes can use message passing
- Any term can be sent as a message

```
ping() ->
    Pid = spawn(fun() -> pong() end),
    Pid ! {ping, self()},
    receive pong -> ok;
       after 1000 -> timeout
    end.
pong() ->
    receive
       {ping, From} -> From ! pong
    end.
```

Refactoring

- Change the structure of the code
  - Rename a variable/function/module...
  - Extract function, inline function
  - Pull up method
- Goals
  - Increase quality
  - Prepare for further development
    or for subsequent transformations
- Preserve semantics

Refactoring tools

- Refactoring is hard (error-prone) to to by hand
  - Simultaneous changes (often global)
  - Conditions for admissibility
- Tool support
- Mostly for OOP
- Less work on FP
  - Haskell (HaRe, Univ. Kent)
  - Clean (prototype, ELU)
  - Erlang (ELU, Univ. Kent)

Refactoring in Erlang

- Set of transformations differs from that for OOP
- FP, so no(t much) side effects
- Not much statically available information
  - Dynamically typed
  - Rules (e.g. scoping) in RM are based on operational semantics
- Higher-order functions
- Reflective programs
- Communication

How to do

- Reformulate the semantic rules
- Perform static analysis
- Assume guidelines and conventions
  - Refactor well-written programs
  - Focus on Erlang/OTP
- Specify clearly the capabilities

Implemented transformations

- Rename variable
- Rename function
- Merge subexpression duplicates
- Eliminate variable
- Extract function
- Reorder function arguments
- Tuple function arguments
- *Currently working on: Introduce records*

Rename variable

- Affects a single function clause
- Variable scoping rules

```
sum_pos(X) ->
  sum(lists:filter(fun(X) -> X>0 end, X)).

sum_pos(L) ->
  sum(lists:filter(fun(X) -> X>0 end, L)).
```

Rename function

- Exported functions called from other modules
- Higher-order functions, reflective calls

```
max(A,B,C) -> max({ A, max({B,C}) }).
max({A,B}) -> if A>B -> A; true -> B end.
```

```
max(A,B,C) -> maxt({ A, maxt({B,C}) }).
maxt({A,B}) -> if A>B -> A; true -> B end.
```

Merge subexpression duplicates

- Finding occurrences of the same expression
- Variables should have the same bindings
- No side-effects

```
sqrt((X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2))
```

```
DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY)
```

Eliminate variable

- Inverse of “Merge subexpression duplicates”
- Unique binding at every elimination position
- Each variable occurring in the bound term is visible at every elimination position
- No side-effects

```
DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY)
```

```
sqrt((X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2))
```

Extract function

- Variables with possible binding occurrences should not appear outside of extracted sequence of expression

```
gcd(X,X) -> X;
gcd(X,Y) -> {A,B} = if X > Y -> {X,Y};
                       true  -> {Y,X}
                    end,
            gcd(A-B,B).
```

```
sort(X,Y) = if X>Y -> {X,Y}; true->{Y,X} end.
gcd(X,X) -> X;
gcd(X,Y) -> {A,B} = sort(X,Y), gcd(A-B,B).
```

Reorder function arguments

```
perimeter({X1,Y1},{X2,Y2},{X3,Y3}) ->
    dist(X1,X2,Y1,Y2) +
    dist(X1,X3,Y1,Y3) +
    dist(X3,X2,Y3,Y2).

dist(X1,X2,Y1,Y2) ->
    DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

```
perimeter({X1,Y1},{X2,Y2},{X3,Y3}) ->
    dist(X1,Y1,X2,Y2) +
    dist(X1,Y1,X3,Y3) +
    dist(X3,Y3,X2,Y2).

dist(X1,Y1,X2,Y2) ->
    DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

Tuple function arguments

```
perimeter({X1,Y1},{X2,Y2},{X3,Y3}) ->
    dist(X1,Y1,X2,Y2) +
    dist(X1,Y1,X3,Y3) +
    dist(X3,Y3,X2,Y2).

dist(X1,Y1,X2,Y2) ->
    DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

```
perimeter({X1,Y1},{X2,Y2},{X3,Y3}) ->
    dist({X1,Y1},{X2,Y2}) +
    dist({X1,Y1},{X3,Y3}) +
    dist({X3,Y3},{X2,Y2}).

dist({X1,Y1},{X2,Y2}) ->    % signature change
    DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

Introduce record

```
perimeter({X1,Y1},{X2,Y2},{X3,Y3}) ->
    dist({X1,Y1},{X2,Y2}) +
    dist({X1,Y1},{X3,Y3}) +
    dist({X3,Y3},{X2,Y2}).

dist({X1,Y1},{X2,Y2}) ->
     DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

```
-record(point,[x,y]).

perimeter(P1,P2,P3) ->
    dist(P1,P2) + dist(P1,P3) + dist(P3,P2).

dist(#point{x=X1,y=Y1},#point{x=X2,y=Y2}) ->
    DX=X1-X2, DY=Y1-Y2, sqrt(DX*DX+DY*DY).
```

Chat server

- Client-server application
- Distributed
- Follow the slides!

```
-module(srv).
-export([start/1, stop/0]).
-export([connect/2, disconnect/1, send/2]).
-define(CLIENT, cli).

% ------------------- Interface functions
% --------------------- Client interface
% ---------------- Implementation functions
```

Interface functions

```
-module(srv).
-export([start/1, stop/0]).
-export([connect/2, disconnect/1, send/2]).
-define(CLIENT, cli).

% ------------------- Interface functions

start(Max) ->
 register(chatsrv, spawn(fun()->init(Max) end)).

stop() -> chatsrv ! stop.
```

Client interface

```
connect(Srv, Nick) ->
    {chatsrv, Srv} ! {connect, self(), Nick},
    receive
      ok -> ok;
      deny -> deny
      after 1000 -> timeout
    end.

disconnect(Srv) ->
    {chatsrv, Srv} ! {disconnect, self()}.

send(Srv, Text) ->
    {chatsrv, Srv} ! {text, self(), Text}.
```

Implementation functions – 1

```
init(Max) -> loop([], Max, 0).

loop(Users, Max, Nr) -> ...

send(_, _, []) -> ok;
send(Msg, Nr, [Pid | Users]) ->
  (?CLIENT):send(Pid,
     "[" ++ (integer_to_list(Nr) ++
             ("]" ++ Msg)) ),
  send(Msg, Nr, Users).
```

```
loop(Users, Max, Nr) ->
 receive
   stop -> ok;
   {connect, Pid, Nick} ->
       if length(Users) >= Max ->
                  Pid ! deny, loop(Users, Max, Nr);
          true -> {link(Pid),Pid} ! ok,
                  NewUsers = [{Nick, Pid} | Users],
                  loop(NewUsers, Max, Nr)
       end;
   {text, From, Text} ->
       [{Nick, _}] =
        lists:filter(fun({_, Pid}) -> Pid == From end,Users),
       Msg = Nick ++ (": " ++ Text),
       send(Msg, Nr + 1, Users),
       loop(Users, Max, Nr + 1);
   {disconnect, Pid} ->
       NewUsers = lists:filter(fun({_, P}) -> P /= Pid end,
                               Users),
       loop(NewUsers, Max, Nr)
 end.
```

Erlang refactorer

- Univ. Kent: traversal of annotated AST
- ELU
  - Graph representation of programs
  - Semantic information is represented as labeled edges
  - Graphs are stored in DB
  - Transformations implemented using SQL
  - User interface: integrated into Emacs
- Cooperation

Conclusions

- Refactoring is useful for industry
- Refactoring in Erlang is difficult, but possible
- Tool being developped by ELU and Univ. Kent
- Seven transformations are already implemented
- Try them out at the lab!