Introducing Records by Refactoring[1]

László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király

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

Sixth ACM SIGPLAN Erlang Workshop
Freiburg, Germany, October 5, 2007

[1] Supported by GVOP-3.2.2-2004-07-0005/3.0 (ELTE IKKK) and Ericsson Hungary and ELTE CNL and OMAA-ÖAU 66öu2.

Contents

Refactoring

- Semantics preserving transformations of source code
  - Rename a variable/function/module. . .
  - Extract function, inline function
  - Turn tuple into record
- Goals
  - Increase quality
  - Prepare for further development
    or for subsequent transformations

Example: Tuple to record

```
init(Time) -> loop({Time, empty(), empty()}).
loop({Time, P, OP}) ->
  receive
    {next} -> do_next(Time,P,OP);
    {get,From,Key} -> do_get(Time,P,OP,From,Key);
    {set,Key,Value} -> do_set(Time,P,OP,Key,Value)
  end.
```

```
-record(state,time,pstore,opstore).
init(Time) -> loop(#state{time=Time,
                          pstore=empty(),
                          opstore=empty()}
                  ).
loop(#state{time=Time,pstore=P,opstore=OP}) ->
  receive
    {next} -> do_next(Time,P,OP);
    {get,From,Key} -> do_get(Time,P,OP,From,Key);
    {set,Key,Value} -> do_set(Time,P,OP,Key,Value)
  end.
```

Refactoring tools

- Cumbersome & error-prone to do by hand
  - Many simultaneous changes
  - Conditions for admissibility
- Tool support
- Mostly for OOP
- Less work on FP
  - Haskell (HaRe, Univ. Kent)
  - Clean (prototype, ELU)
  - Erlang: cooperation between UK/UK and ELU/HU
    - Wrangler (UK)
    - RefactorErl (ELU)

Refactoring in Erlang

- Set of transformations differs from that for OOP
- Things that help
  - FP: referential transparency
  - Assume conventions and guidelines (OTP)
- Things that hurt
  - Side effects
  - Higher-order functions
  - Reflective programs
  - Communication
  - Dynamic typing
  - Lack of programmer defined types

Preserving semantics

Principle 1

Refactorings should not change the meaning of the program.

- The tool is shy
- Too restrictive in practice

```
-module(a).
-export([f/0,egg/0]).
factor(X) -> ... % prime factorization
egg() -> 42.
f() -> apply( list_to_atom(factor(97)),
              list_to_atom(factor(1071509)),
              []
            ).
```

- Instead: specify properly the limitations

Introducing records

- Request from industry
- Turn tuples into records
  - Records correspond to programmer defined types (increased safety and readability)
  - Records provide a flexible structure for further development
- Changing a single tuple is not enough
- Basic transformation + propagation
- The topic of this talk: design of propagation

Case study: time-based property server

```
init(Time) -> loop({Time, empty(), empty()}).
loop(State) ->
  receive
    {next}          -> do_next(State);
    {get,From,Key}  -> do_get(State,From,Key);
    {set,Key,Value} -> do_set(State,Key,Value)
  end.
do_next({peak, P, OP}) -> loop({offp, P, OP});
do_next({offp, P, OP}) -> loop({peak, P, OP}).
do_get(State = {peak, P, OP}, From, Key) ->
  get_value(From, P, Key),
  loop(State);
do_get(State = {offp, P, OP}, From, Key) -> ...
do_set({peak, P, OP}, Key, Value) ->
  NewSt = set_value(P, Key, Value),
  loop({peak, NewSt, OP});
do_set({offp, P, OP}, Key, Value) -> ...
```

Basic transformation

```
loop(State) ->
  receive
    {next} -> do_next(State);
    {get,From,Key}  -> do_get(State,From,Key);
    {set,Key,Value} -> do_set(State,Key,Value)
  end.

do_next({peak, P, OP}) ->
  loop({offp, P, OP});
do_next({offp, P, OP}) ->
  loop({peak, P, OP}).
```

Basic transformation

```
-record(state,time,pstore,opstore).
tuple_to_state({E1,E2,E3}) ->
  #state{time=E1,pstore=E2,opstore=E3};
tuple_to_state(E) -> E.

loop(State) ->
  receive
    {next} -> do_next(tuple_to_state(State));
    {get,From,Key}  -> do_get(State,From,Key);
    {set,Key,Value} -> do_set(State,Key,Value)
  end.

do_next(#state{time=peak, pstore=P, opstore=OP}) ->
  loop({offp, P, OP});
do_next(#state{time=offp, pstore=P, opstore=OP}) ->
  loop({peak, P, OP});
```

Applying the basic transformation again

```
loop(State) ->
  receive
    {next} -> do_next(tuple_to_state(State));
    {get,From,Key}
             -> do_get(State,From,Key);
    {set,Key,Value}
             -> do_set(State,Key,Value)
  end.

do_next(#state{time=peak,pstore=P,opstore=OP}) -> ...

do_get(State={peak,P,OP},From,Key) ->
  get_value(From, P, Key),
  loop(State);
do_get(State,From,Key) ->
  get_value(From,element(3,State),Key),
  loop(State).
```

Applying the basic transformation again

```
loop(State) ->
  receive
    {next} -> do_next(tuple_to_state(State));
    {get,From,Key}
             -> do_get(tuple_to_state(State),From,Key);
    {set,Key,Value}
             -> do_set(State,Key,Value)
  end.

do_next(#state{time=peak,pstore=P,opstore=OP}) -> ...

do_get(State=#state{time=peak,pstore=P,opstore=OP},From,
  get_value(From, P, Key),
  loop(state_to_tuple(State));
do_get(State,From,Key) ->
  get_value(From,element(3,state_to_tuple(State)),Key),
  loop(state_to_tuple(State)).
```

Applying the basic transformation again

```
loop(State) ->
  receive
    {next} -> do_next(tuple_to_state(State));
    {get,From,Key}
             -> do_get(tuple_to_state(State),From,Key);
    {set,Key,Value}
             -> do_set(State,Key,Value)
  end.

do_next(#state{time=peak,pstore=P,opstore=OP}) -> ...

do_get(State=#state{time=peak,pstore=P,opstore=OP},From,
  get_value(From, P, Key),
  loop(state_to_tuple(State));
do_get(State,From,Key) ->
  get_value(From,State#state.opstore,Key),
  loop(state_to_tuple(State)).
```

Principles

Principle 1

The refactoring should not change the meaning of the program.

Principle 2

The refactoring should transform everything that the programmer wants to change.

Principle 3

The refactoring should not transform anything that the programmer wants to remain intact.

Principles

Principle 1

The refactoring should not change the meaning of the program.

Principle 2

The refactoring should transform everything that the programmer wants to change.

Principle 3

The refactoring should not transform anything that the programmer wants to remain intact.

Principles

Principle 1

The refactoring should not change the meaning of the program.

Principle 2

The refactoring should transform everything that the programmer wants to change.

Principle 3

The refactoring should not transform anything that the programmer wants to remain intact.

Introduce records: iteration of basic transformations

Preparation

- Reorder function arguments
- Tuple function arguments

Introduce records

- Select a tuple skeleton, provide record name and field names
- Convert directly affected expressions (basic transf.)
- Find and convert derived expressions (propagation)
- Introduce record updates

Introduce records: iteration of basic transformations

Preparation

- Reorder function arguments
- Tuple function arguments

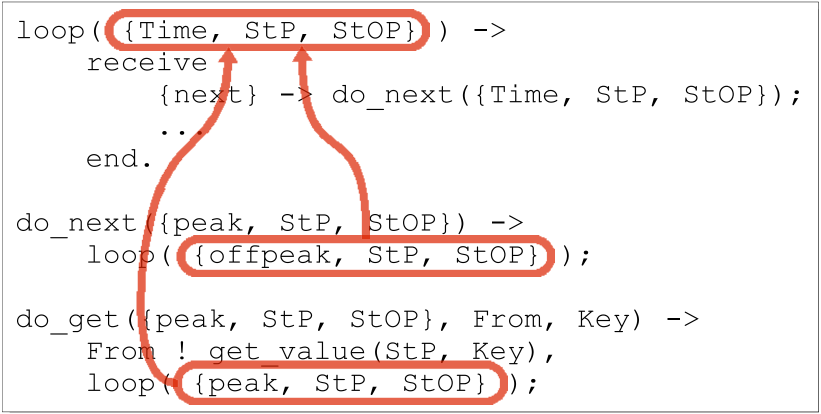
Introduce records

- Select a tuple skeleton,
  provide record name and field names
- Convert directly affected expressions (basic transf.)
- Find and convert derived expressions (propagation)
- Introduce record updates

Preparation

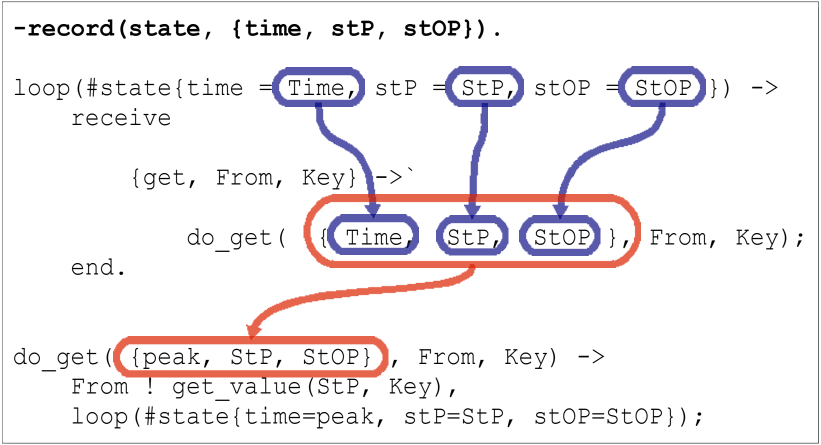
```
loop(Time, StP, StOP) ->
    receive
        {next}            -> do_next(Time, StP, StOP);
        {get, From, Key}  -> do_get(Time, StP, StOP, From, Key);
        {set, Key, Value} -> do_set(Time, StP, StOP, Key, Value)
    end.

do_next(peak,    StP, StOP) -> loop(offpeak, StP, StOP);
do_next(offpeak, StP, StOP) -> loop(peak, StP, StOP).

do_get(peak, StP, StOP, From, Key) ->
    From ! get_value(StP, Key),  loop(peak, StP, StOP);
do_get(offpeak, StP, StOP, From, Key) ->
    From ! get_value(StOP, Key), loop(offpeak, StP, StOP).

do_set(peak, StP, StOP, Key, Value) ->
    NewSt = set_value(StP, Key, Value),  loop(peak, NewSt, StOP);
do_set(offpeak, StP, StOP, Key, Value) ->
    NewSt = set_value(StOP, Key, Value), loop(offpeak, StP, NewSt).
```

Convert directly affected expressions



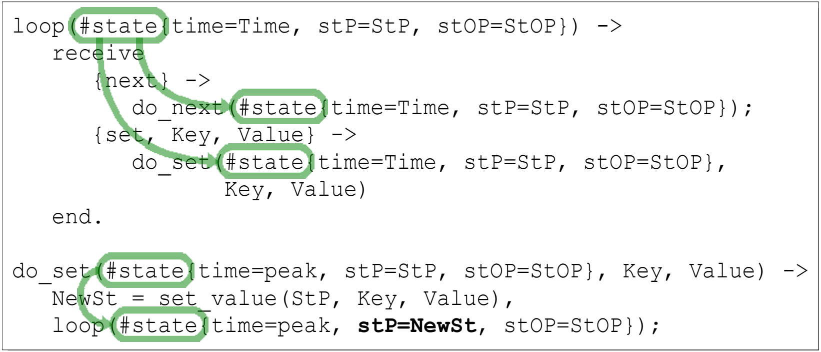
```
loop({Time, StP, StOP}) ->
    receive
        {next} -> do_next({Time, StP, StOP});
        ...
    end.

do_next({peak, StP, StOP}) ->
    loop({offpeak, StP, StOP});

do_get({peak, StP, StOP}, From, Key) ->
    From ! get_value(StP, Key),
    loop({peak, StP, StOP});
```

Find and convert derived expressions



```
-record(state, {time, stP, stOP}).

loop(#state{time = Time, stP = StP, stOP = StOP}) ->
    receive

        {get, From, Key} ->`

            do_get( {Time, StP, StOP }, From, Key);
    end.


do_get({peak, StP, StOP}, From, Key) ->
    From ! get_value(StP, Key),
    loop(#state{time=peak, stP=StP, stOP=StOP});
```

Introducing record updates



```
loop(#state{time=Time, stP=StP, stOP=StOP}) ->
   receive
      {next} ->
         do_next(#state{time=Time, stP=StP, stOP=StOP});
      {set, Key, Value} ->
         do_set(#state{time=Time, stP=StP, stOP=StOP},
                Key, Value)
   end.

do_set(#state{time=peak, stP=StP, stOP=StOP}, Key, Value) ->
   NewSt = set_value(StP, Key, Value),
   loop(#state{time=peak, stP=NewSt, stOP=StOP});
```

Refactored code

```
-record(state, {time, stP, stOP}).
loop(St=#state{}) ->
    receive
        {next}            -> do_next(St);
        {get, From, Key}  -> do_get(St, From, Key);
        {set, Key, Value} -> do_set(St, Key, Value)
    end.

do_next(St=#state{time=peak}) ->
    loop(St#state{time=offpeak});
do_next(St=#state{time=offpeak}) ->
    loop(St#state{time=peak}).

do_get(St=#state{time=peak, stP=StP}, From, Key) ->
    From ! get_value(StP, Key),     loop(St);
do_get(St=#state{time=offpeak, stOP=StOP}, From, Key) ->
    From ! get_value(StOP, Key),    loop(St).

do_set(St=#state{time=peak, stP=StP}, Key, Value) ->
    NewSt = set_value(StP, Key, Value),
    loop(St#state{stP=NewSt});
do_set(St=#state{time=offpeak, stOP=StOP}, Key, Value) ->
    NewSt = set_value(StOP, Key, Value),
    loop(St#state{stOP=NewSt}).`
```

Transformation rules

- Given a tuple skeleton, turn tuple constructor to record constructor
- Given a tuple pattern, transform all expressions matched against it
- Given a tuple expression, transform all patterns matched against it
- Propagate records
  - Through variable
  - Through compound expression (block, branch, function call)
- Propagate fields
- Apply converters when everything else fails

Make the refactorer tool more interactive?

RefactorErl

- **http://plc.inf.elte.hu/erlang/**
- Released under EPL
- Cool installer
- Windows, Linux, Mac, (Solaris)
- Current version: 0.1.1
- 8 transformations
- New major release is planned in 2008
- Happy to show you!

Implemented transformations

- Rename variable
- Rename function
- Merge subexpression duplicates
- Eliminate variable
- Extract function
- Reorder function arguments
- Tuple function arguments
- *Preliminary/partial implementation: Introduce records*

The implementation of the tool

- Written in Erlang and SQL
- User interface: Emacs and Distel
- Front-end: `epp_dodger`, hacked `erl_scan` and `erl_recomment`
- Output: standard pretty-printer
- Inside: AST extended into semantic graph
- Back-end: MySQL database

Future work: new major release of RefactorErl

- Preserve layout (own scanner/parser/pp)
- Support for macros
- Improved performance (mnesia)
- Undo facility
- More transformations

Conclusions

- RefactorErl: $7 + \varepsilon$ transformations already implemented
- Design of “Introduce record”
- Three principles
- Basic transformation not enough: *propagation*
- Feedback appreciated!