

# Automated syntax manipulation in RefactorErl

## Progress of the RefactorErl Project <sup>1</sup>

R. Kitlei L. Lövei M. Tóth Z. Horváth T. Kozsik  
R. Király I. Bozó Cs. Hoch D. Horpácsi



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

14th International Erlang/OTP User Conference, 2008

<sup>1</sup> Supported by ELTE IKKK and Ericsson Hungary.

# Outline

- 1 Introduction
  - RefactorErl
- 2 Subtree construction
- 3 Subtree replacement or insertion
- 4 Implemented transformations
  - An example

# RefactorErl, an Erlang Refactoring Tool

- Refactoring is meaning-preserving source code transformation
- RefactorErl is a tool that refactors Erlang source code
- Two type of refactorings:
  - Do not change the shape of the syntax tree, only update the information
  - Construct, delete, move, insert or replace subtrees in the syntax tree

# Construct new elements

- Use the parser to generate the subtree from source code fragments or
- Do it by hand or
- New mechanism: generate from the grammar description by adding the content and the description of the node
  - Safe - higher abstraction level
  - Reusable existing subcomponents (Compositional)
  - Structure of the graph and layout of unaffected subcomponents preserved
  - Legibility
  - Easy to maintain with respect to grammar changes

# Create a new application node

```
— Construction of a function application node —  
Name = referl_syntax:create(  
    #expr{kind = atom},  
    [atom_to_list(AppName)] ),  
Args = [referl_syntax:create(  
    #expr{kind = variable},  
    [AName] ) || AName <- AppArgNames],  
App   = referl_syntax:create(  
    #expr{kind = application},  
    [{sub, [Name]}, {sub, Args}] ),
```

## Insert or replace nodes

- The new nodes to be inserted and the place of insertion or replacement have to be specified.
  - Replacement of all or part of nodes of the same type.
  - Replacement of a range of nodes.
  - Insertion before or after a node.
- Determines the affected part in the syntax tree.
- Makes the change.
- Checks consistency with grammar and guarantees the resulting structure conforms to the grammar description.

## Replace the selection with the application

Construction of a function application node

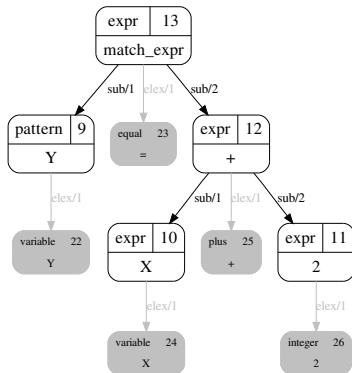
```
referl_syntax:replace(  
  Parent,  
  {From, To},  
  [{body, [App]}])
```

Construction of a function application node

```
referl_syntax:replace(  
  Parent,  
  {node, SelectedNode},  
  [{sub, [App]}])
```

# The representation of the selected expression

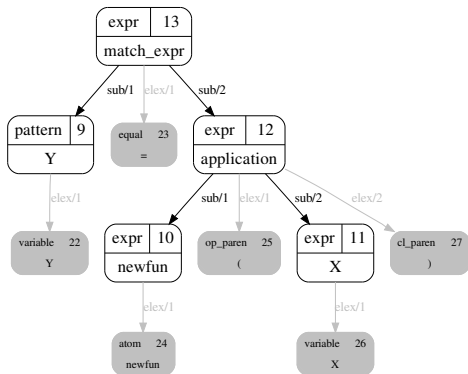
```
extract.erl  
f(X) ->  
  Y=X+2.
```





# The representation of the function application that replaces the selection

```
extract.erl  
f(X) ->  
    Y = newfun(X).  
  
newfun(X) ->  
    X + 2.
```



# Transformations

- *Extract function* - LOC of tree construction 20 vs 310 ( + Side condition analysis - 420 LOC)

## Reimplemented

- *Move function definition*
- *Inline function*
- *Reorder function arguments*
- *Tuple function arguments*
- *Rename variable*
- *Rename function*
- *Eliminate variable*
- *Merge subexpression duplicates*

## New transformations

- *Generalize function*
- *Rename module*
- *Rename record*
- *Rename record field*
- *Move record*
- *Expand funexpression*
- *(Module clustering)*

## Eliminate variable Lst in sum\_n

```
-module (demo) .  
-export ([sum_n/1]).  
-include ("global.hrl").  
  
sum([]) ->  
    0;  
sum([H|T]) ->  
    S = sum(T),  
    H + S.  
  
sum_n(N) ->  
    Lst = lists:seq(?START, N),  
    sum(Lst).
```

## Lst is eliminated

```
-module (demo) .  
-export ([sum_n/1]).  
-include ("global.hrl").  
  
sum([]) ->  
    0;  
sum([H|T]) ->  
    S = sum(T),  
    H + S.  
  
sum_n(N) ->  
    sum((lists:seq(?START, N))).
```

## Generalize function sum by 0 in the first clause

```
-module(demo) .  
-export([sum_n/1]) .  
-include("global.hrl") .  
  
sum([]) ->  
    0;  
sum([H|T]) ->  
    S = sum(T),  
    H + S.  
  
sum_n(N) ->  
    sum((lists:seq(?START, N))).
```

## sum/1 is generalized

```
-module(demo) .  
-export([sum_n/1]) .  
-include("global.hrl") .  
  
sum([], Z) ->  
    Z;  
sum([H|T], Z) ->  
    S = sum(T, Z),  
    H + S.  
  
sum_n(N) ->  
    sum((lists:seq(?START,N)), 0) .
```

## Extract H+S from sum with the name plus

```
-module(demo) .  
-export([sum_n/1]) .  
-include("global.hrl") .  
  
sum([], Z) ->  
    Z;  
sum([H|T], Z) ->  
    S = sum(T, Z),  
    H + S.  
  
sum_n(N) ->  
    sum((lists:seq(?START,N)), 0) .
```

## H + S extracted

```
-module(demo) .  
-export([sum_n/1]).  
-include("global.hrl").
```

```
sum([], Z) ->  
    Z;  
sum([H|T], Z) ->  
    S = sum(T, Z),  
    plus(H, S).
```

```
plus(H, S) ->  
    H + S.
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)), 0).
```



## Generalize function sum by plus(H,S)

```
-module (demo) .  
-export ([sum_n/1]).  
-include ("global.hrl").
```

```
sum([], Z) ->  
    Z;  
sum([H|T], Z) ->  
    S = sum(T, Z),  
    plus(H, S).
```

```
plus(H, S) -> H + S.
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)), 0).
```

## sum/2 is generalized

```
-module(demo) .  
-export([sum_n/1]).  
-include("global.hrl").  
  
sum([], Z, _Plus) ->  
    Z;  
sum([H|T], Z, Plus) ->  
    S = sum(T, Z, Plus),  
    Plus(H, S).  
  
plus(H, S) -> H + S.  
  
sum_n(N) ->  
    sum((lists:seq(?START, N)), 0,  
        fun(H,S) -> plus(H, S) end).
```

## Inline function call plus(H,S) in sum\_n

```
-module(demo) .  
-export([sum_n/1]).  
-include("global.hrl").
```

```
sum([], Z, _Plus) ->  
    Z;  
sum([H|T], Z, Plus) ->  
    S = sum(T, Z, Plus),  
    Plus(H, S).
```

```
plus(H, S) -> H + S.
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)), 0,  
        fun(H,S) -> plus(H, S) end).
```

## plus(H,S) is inlined

```
-module(demo) .  
-export([sum_n/1]).  
-include("global.hrl").
```

```
sum([], Z, _Plus) ->  
    Z;  
sum([H|T], Z, Plus) ->  
    S = sum(T, Z, Plus),  
    Plus( H, S).
```

```
plus(H, S) -> H + S.
```

```
sum_n(N) ->  
    sum((lists:seq(?START,N)), 0,  
        fun(H,S) -> H + S end).
```

## Reverse the order of the last two arguments of sum

```
-module(demo) .  
-export([sum_n/1]).  
-include("global.hrl").
```

```
sum([], Z, _Plus) ->  
    Z;  
sum([H|T], Z, Plus) ->  
    S = sum(T, Z, Plus),  
    Plus( H, S).
```

```
plus(H, S) -> H + S.
```

```
sum_n(N) ->  
    sum((lists:seq(?START,N)), 0,  
        fun(H,S) -> H + S end).
```

## The order of the last two arguments is changed

```
-module(demo) .  
-export([sum_n/1]).  
-include("global.hrl").  
  
sum([], _Plus, Z) ->  
    Z;  
sum([ H | T ], Plus, Z) ->  
    S = sum(T, Plus, Z),  
    Plus(H, S).  
  
plus(H, S) -> H + S.  
  
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H,S) -> H + S end, 0).
```

## Move record cplx from cplx.erl to global.hrl

```
-module(cplx) .  
-export([add/2]) .  
-record(cplx, {re=0.0, im=0.0}) .  
  
add(#cplx{re=Re1, im=Im1}, #cplx{re=Re2, im=Im2}) ->  
    #cplx{re=Re1+Re2, im=Im1+Im2} .  
  
-----  
  
-define(START, 1) .
```

## The record cplx moved to global.hrl

```
-module(cplx).  
-export([add/2]).  
-include("/home/melinda/Documents/global.hrl").  
  
add(#cplx{re=Re1, im=Im1}, #cplx{re=Re2, im=Im2}) ->  
    #cplx{re=Re1+Re2, im=Im1+Im2}.  
  
-----  
  
-define(START, 1).  
  
-record(cplx, {re = 0.0, im = 0.0}).
```



## Add a new function to demo.erl

```
-include(global.hrl).
```

```
sum([], _Plus, Z) ->  
    Z;
```

```
sum([ H | T], Plus, Z) ->  
    S = sum( T, Plus, Z),  
    Plus(H, S).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H,S) -> H + S end, 0).
```

```
sum_cplx(Lst) ->  
    sum(Lst, fun cplx:add/2, #cplx{}).
```

## Rename record cplx to complex

```
-define (START, 1) .
```

```
-record(cplx, {re = 0.0, im = 0.0}).
```

---

```
-module(cplx) .
```

```
-export([add/2]) .
```

```
-include("/home/melinda/Documents/global.hrl").
```

```
add(#cplx{re=Re1, im=Im1}, #cplx{re=Re2, im=Im2}) ->  
    #cplx{re=Re1+Re2, im=Im1+Im2}.
```

## cplx is renamed to complex

```
-define (START, 1) .  
  
-record (complex, {re = 0.0, im = 0.0}).  
  
-----  
  
-module (cplx) .  
-export ([add/2]) .  
-include ("/home/melinda/Documents/global.hrl") .  
  
add (#complex {re=Re1, im=Im1},  
    #complex {re=Re2, im=Im2}) ->  
    #complex {re=Re1+Re2, im=Im1+Im2} .
```

## demo.erl is also changed

```
-include(global.hrl).
```

```
sum([], _Plus, Z) ->  
    Z;
```

```
sum([ H | T ], Plus, Z) ->  
    S = sum( T, Plus, Z ),  
    Plus( H, S ).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H,S) -> H + S end, 0).
```

```
sum_cplx(Lst) ->  
    sum(Lst, fun cplx:add/2, #complex{}).
```

## Rename module `cplx` to `complex`

```
-module(cplx).  
-export([add/2]).  
-include("/home/melinda/Documents/global.hrl").  
  
add(#complex{re=Re1, im=Im1},  
    #complex{re=Re2, im=Im2}) ->  
    #complex{re=Re1+Re2, im=Im1+Im2}.  
  
-----  
  
sum_cplx(Lst) ->  
    sum(Lst, fun cplx:add/2, #complex{}).
```

## cplx is renamed to complex

```
-module (complex) .  
-export ([add/2]).  
-include (" /home/melinda/Documents/global.hrl" ) .  
  
add(#complex{re=Re1, im=Im1},  
    #complex{re=Re2, im=Im2}) ->  
    #complex{re=Re1+Re2, im=Im1+Im2} .  
  
-----  
  
sum_cplx(Lst) ->  
    sum(Lst, fun complex:add/2, #complex{}).
```

## Rename record field re to real and im to imag

```
-define (START, 1) .  
  
-record (complex, {re = 0.0, im = 0.0}).  
  
-----  
  
-module (complex) .  
-export ([add/2]) .  
-include (" /home/melinda/Documents/global.hrl" ) .  
  
add (#complex {re=Re1, im=Im1},  
     #complex {re=Re2, im=Im2}) ->  
     #complex {re=Re1+Re2, im=Im1+Im2} .
```

## Record fields are renamed

```
-module (complex) .  
-export ([add/2]).  
-include (" /home/melinda/Documents/global.hrl" ) .  
  
add(#complex{real=Re1, imag=Im1},  
    #complex{real=Re2, imag=Im2}) ->  
    #complex{real=Re1+Re2, imag=Im1+Im2} .
```



## Rename function complex:add/2 to plus

```
-module (complex) .  
-export ([add/2]).  
-include (" /home/melinda/Documents/global.hrl").
```

```
add(#complex{real=Re1, imag=Im1},  
    #complex{real=Re2, imag=Im2}) ->  
    #complex{real=Re1+Re2, imag=Im1+Im2}.
```

---

```
sum_cplx(Lst) ->  
    sum(Lst, fun complex:add/2, #complex{}).
```

## Function is renamed to plus

```
-module (complex) .  
-export ([plus/2]) .  
-include ("/home/melinda/Documents/global.hrl") .
```

```
plus (#complex{real=Re1, imag=Im1},  
      #complex{real=Re2, imag=Im2}) ->  
      #complex{real=Re1+Re2, imag=Im1+Im2}
```

---

```
sum_cplx(Lst) ->  
    sum(Lst, fun complex:plus/2, #complex{}).
```

## Rename variable H in sum to Head and T to Tail

```
-include(global.hrl).
```

```
sum([], _Plus, Z) ->  
    Z;
```

```
sum([H | T], Plus, Z) ->  
    S = sum(T, Plus, Z),  
    Plus(H, S).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H,S) -> H + S end, 0).
```

```
sum_cplx(Lst) ->  
    sum(Lst, fun complex:plus/2, #complex{}).
```

## Variables are renamed

```
-include(global.hrl).
```

```
sum([], _Plus, Z) ->  
    Z;
```

```
sum([ Head | Tail], Plus, Z) ->  
    S = sum(Tail, Plus, Z),  
    Plus(Head, S).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H,S) -> H + S end, 0).
```

```
sum_cplx(Lst) ->  
    sum(Lst, fun complex:plus/2, #complex{}).
```

## Tuple the last two arguments of sum

```
-include(global.hrl).
```

```
sum([], _Plus, Z) ->  
    Z;
```

```
sum([Head | Tail], Plus, Z) ->  
    S = sum(Tail, Plus, Z),  
    Plus(Head, S).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H, S) -> H + S end, 0).
```

```
sum_cplx(Lst) ->  
    sum(Lst, fun complex:plus/2, #complex{}).
```

## The arguments are tupled

```
-include(global.hrl).
```

```
sum([], {_Plus, Z}) ->  
    Z;
```

```
sum([Head | Tail], {Plus, Z}) ->  
    S = sum(Tail, {Plus, Z}),  
    Plus(Head, S).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        {fun(H, S) -> H + S end, 0}).
```

```
sum_cplx(Lst) ->  
    sum(Lst, {fun complex:plus/2, #complex{}}).
```

## Move function `sum_cplx` from `demo.erl` to `complex.erl`

```
-include(global.hrl).
```

```
sum([], _Plus, Z) ->  
    Z;
```

```
sum([Head | Tail], Plus, Z) ->  
    S = sum(Tail, Plus, Z),  
    Plus( Head, S).
```

```
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        {fun(H, S) -> H + S end, 0}).
```

```
sum_cplx(Lst) ->  
    sum(Lst, {fun plus/2, #complex{}}).
```

## The function is moved to complex.erl

```
-module (complex) .  
-export ([plus/2]) .  
-include (" /home/melinda/Documents/global.hrl" ) .  
  
plus (#complex {real=Re1, imag=Im1},  
      #complex {real=Re2, imag=Im2}) ->  
    #complex {real=Re1+Re2, imag=Im1+Im2} .  
  
sum_cplx (Lst) ->  
    demo:sum (Lst, {fun plus/2, #complex {}}) .
```



## The function is moved from demo.erl

```
-export([sum/2]).  
-include(global.hrl).  
  
sum([], _Plus, Z) ->  
    Z;  
sum([Head | Tail], Plus, Z) ->  
    S = sum(Tail, Plus, Z),  
    Plus(Head, S).  
  
sum_n(N) ->  
    sum((lists:seq(?START, N)),  
        fun(H, S) -> H + S end, 0).
```

## Summary

- Enhances the readability and reliability of the invocation of refactorings
- Straightforward use of toolset for implementing the construction part of further refactorings
- Real industrial source code can be parsed and analyzed
- Successfully applied for module clustering and restructuring on industrial software
- 7 common refactorings with Wrangler, possibility for cross-testing
- Complementary functionalities: 5 Wrangler + 7 RefactorErl different transformations and Duplicate code detection(Wrangler) + module clustering(RefactorErl)

<http://plc.inf.elte.hu/erlang>