Refactoring and Analysis with RefactorErl

László Lövei, Melinda Tóth, Zoltán Horváth [1]

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

RefactorErl team:
Tamás Kozsik, Róbert Kitlei, Roland Király, István Bozó

November 13, 2009

[1] Supported by TECH_08_A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary

Outline

History

- Original idea: SQL based refactoring (Clean)
- Research on Erlang refactoring (Ericsson Hungary)
- Experiments
  - MySQL, standard parser and pretty printer
  - Mnesia, custom parser, whitespace preservation
- Real-world applications for analysis

Design goals

1. Store semantic information instead of calculating each time
   - Efficient retrieval – graph model
   - Incremental analysis
2. Provide a platform for source code transformation
   - Generic solutions are preferred
   - Non-refactoring applications

Requirements

- Work with large code base
- Language coverage
- Comment preservation
- Layout preservation (indentation)

Three-layered graph model

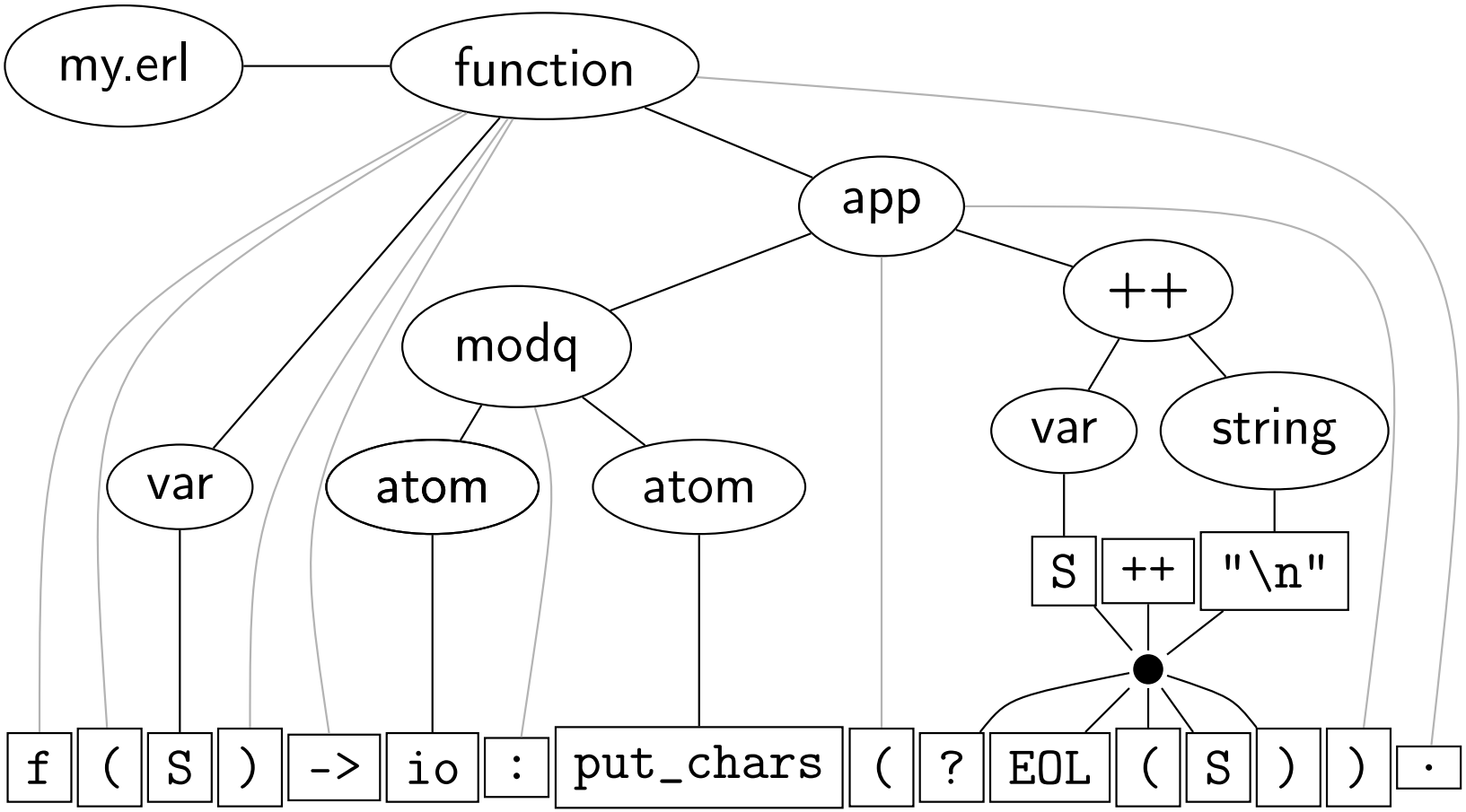
1. Lexical level
   - tokens
   - preprocessing
   - comments, whitespace
2. Syntax level
   - abstract syntax tree
   - files
3. Semantic level
   - module, function, record, variable nodes
   - links to definition and usage

```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

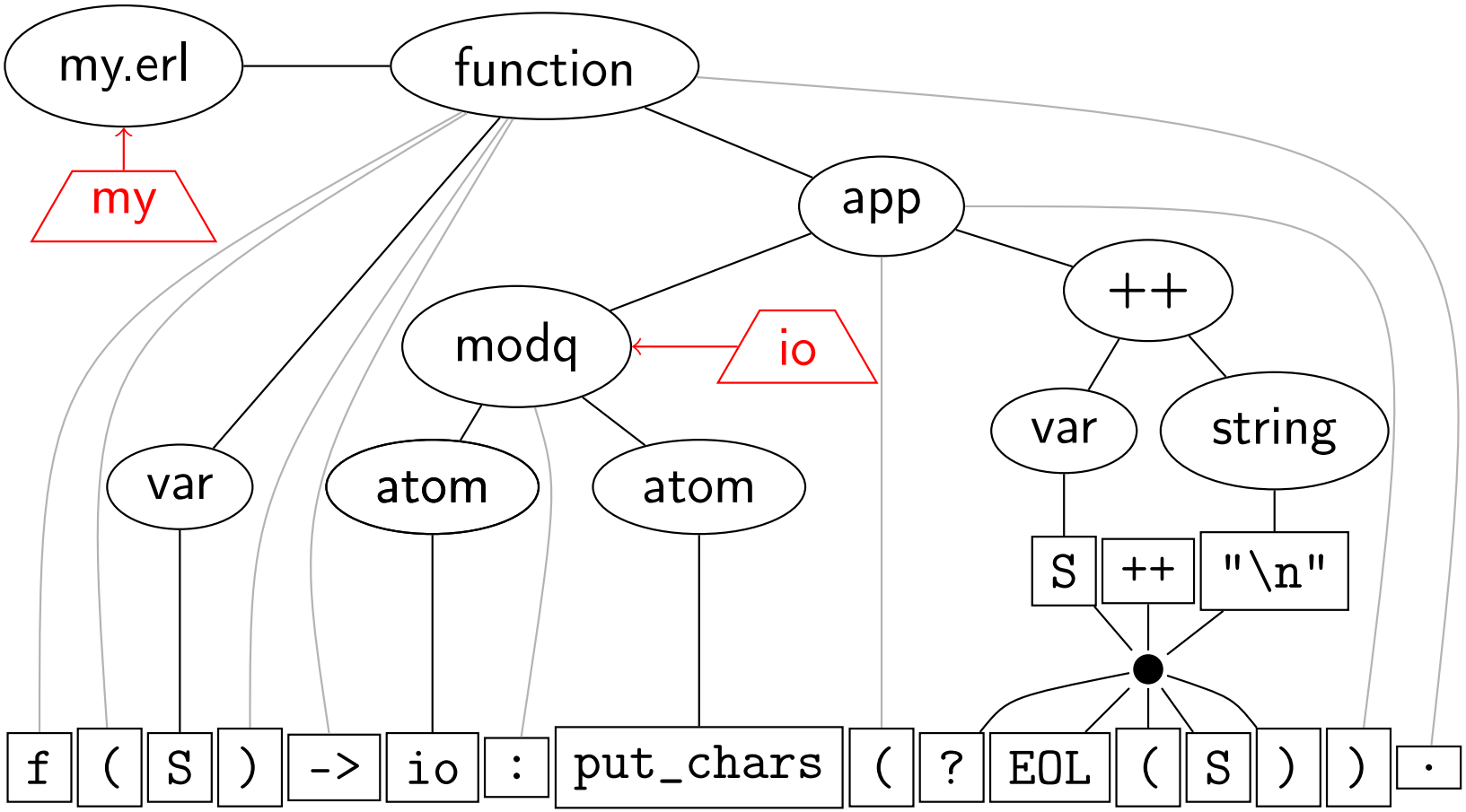
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```

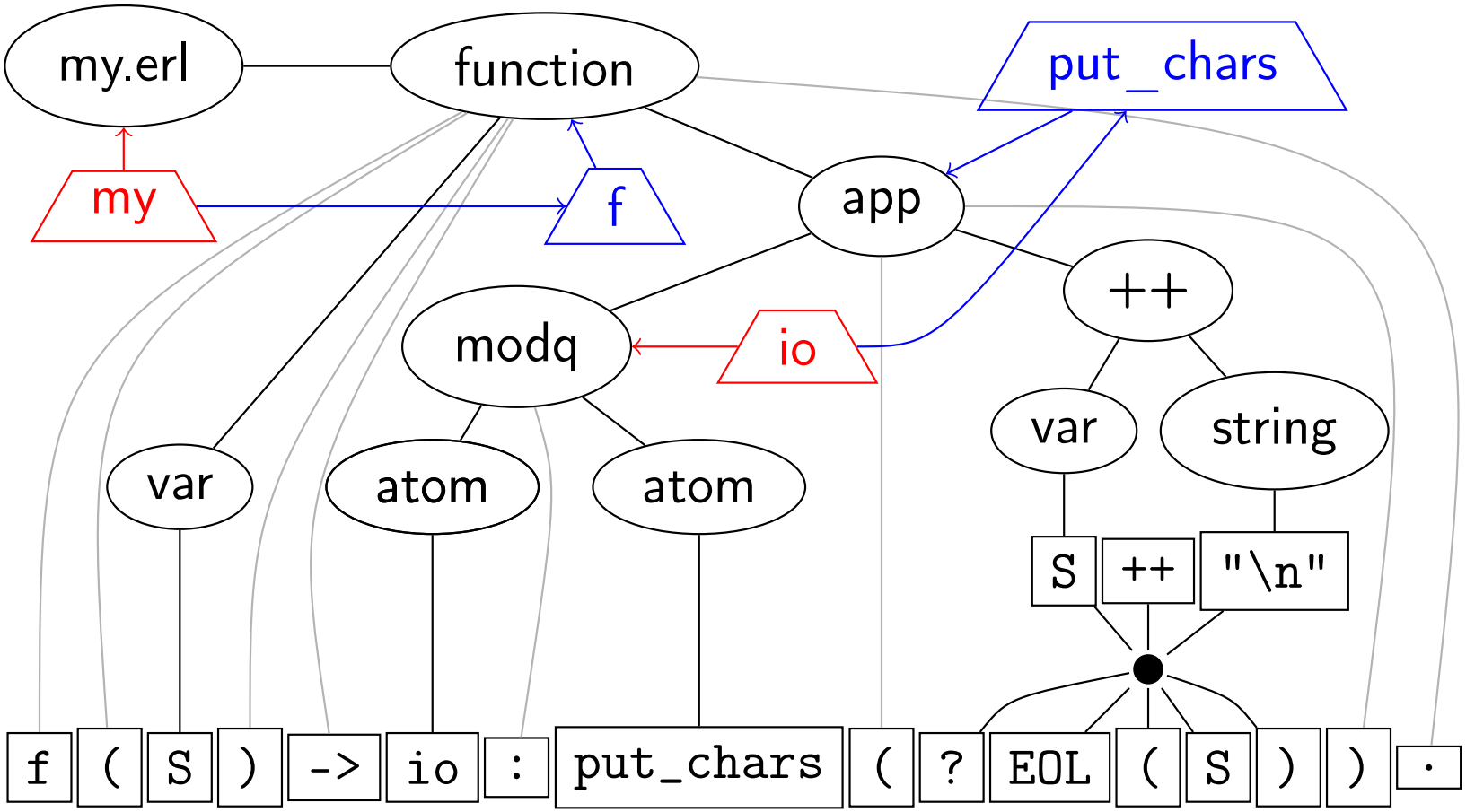
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



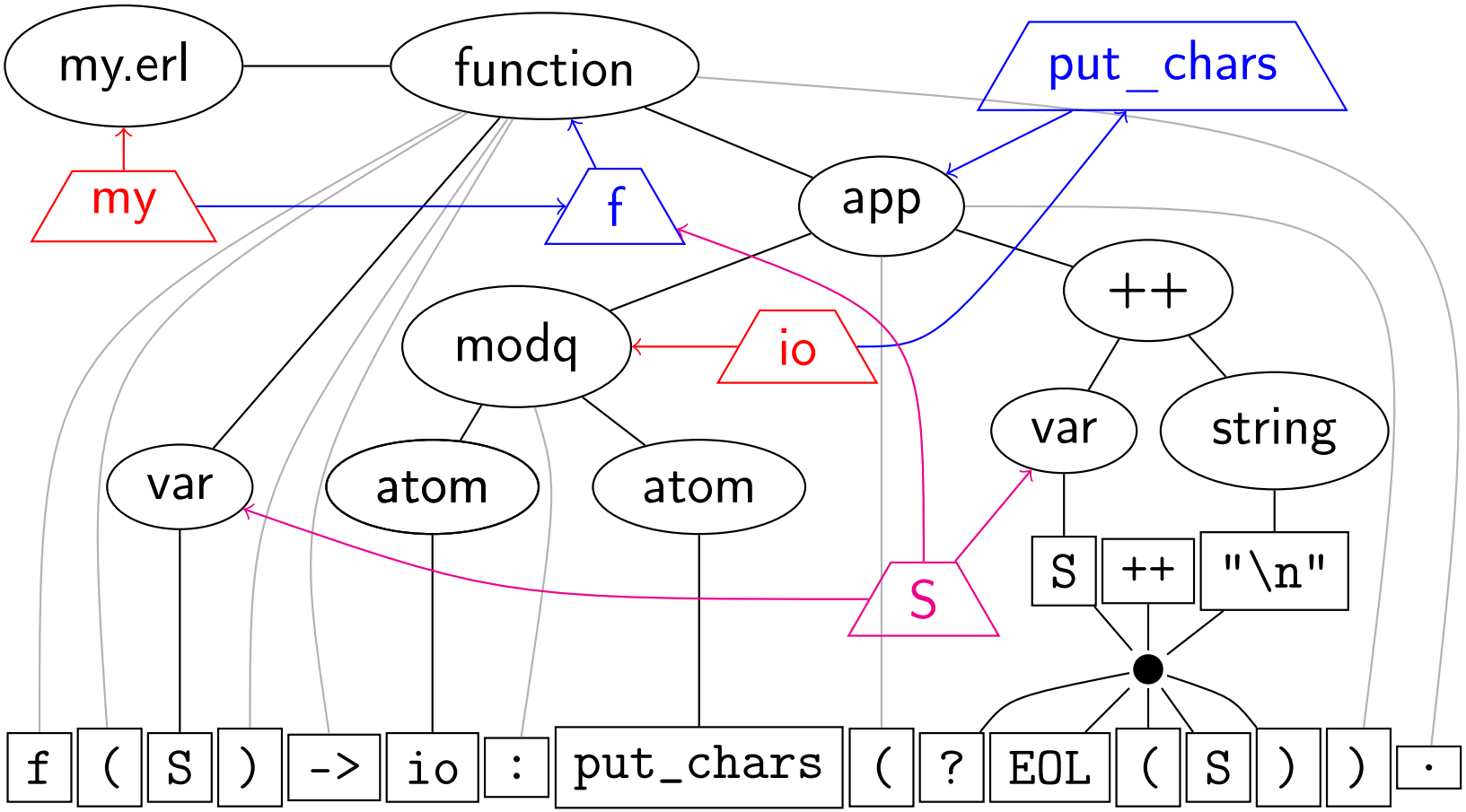
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



Refactoring workflow

1. Read and analyse source code
   - Already finished when refactoring starts
2. Check side conditions
   - Semantic links make it easy and efficient
   - Graph queries simplify graph traversal
3. Apply the transformation
   - Syntax tree based manipulations
4. Save the result
   - Unmodified code is preserved
   - Generated and moved code is pretty printed

Transformation

- Only the syntax tree is manipulated
  - Syntactic nodes can be created or deleted
  - Subtrees can be copied or moved
- Automatic token handling
  - Missing or extra commas and semicolons
  - Generation or removal based on the syntax description
- Automatic analysis
  - Incremental semantic analysis is triggered by syntactic changes
  - Pretty printing is a special kind of analysis

Graph storage

- Nodes and edges are stored in Mnesia tables
  - Node attributes: token text, variable name, ...
  - Edge labels: subexpression, variable reference, ...
- Graph path: filtered edge label sequence
  - Edges are indexed by label
  - Cost doesn't grow with code size
- Frequently used queries need only fixed length paths

Other details

- Extended syntax description
  - Defines the representation
  - Source for parser, lexer, and token updater
- Analyser framework
  - Extensible, modular structure
  - Works on syntactic subtrees (incremental)
- Generic user interface support
  - GNU Emacs, XEmacs
  - ErlIde, Erlang console: on the way

Current limitations

- Dynamic constructs
  - `apply`, `spawn`
  - Message passing
- Type annotations
  - `-type`, `-spec`
- Speed
  - Initial analysis
  - External modifications

Refactoring steps

Rename

- variable
- **function**
- record, record field
- macro
- module
- header file

Move definition

- macro
- **record**
- **function**

Expression structure

- **eliminate variable**
- **merge expressions**
- **extract function**
- **inline function**
- inline macro
- expand fun-expression

Function interface

- **generalize function**
- **reorder parameters**
- tuple parameters
- **introduce import**

Merge expression duplicates

```
-module(compute_pi).

-export([by_madhava/1]).

by_madhava(K) ->
    4*madhava(lists:seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    V = ((-H rem 4)+2)/H,
    V+madhava(T).
```

Expression merged

```
-module(compute_pi).

-export([by_madhava/1]).
by_madhava(K) ->
   4*madhava(lists:seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    V = (( -H rem 4)+2)/H,
    L = madhava(T),
    V+L.
```

Eliminate variable

```
-module(compute_pi).

-export([by_madhava/1]).

by_madhava(K) ->
   4*madhava(lists:seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    V = (( -H rem 4)+2)/H,
    L = madhava(T),
    V+L .
```

V eliminated

```
-module(compute_pi).

-export([by_madhava/1]).

by_madhava(K) ->
   4*madhava(lists:seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    L = madhava(T),
    ((( -H rem 4)+2)/H)+L.
```

Introduce import

```
-module(compute_pi).

-export([by_madhava/1]).

by_madhava(K) ->
    4*madhava(lists:seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    L = madhava(T),
    ((( -H rem 4)+2)/H)+L.
```

New import added

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    L = madhava(T),
    ((( -H rem 4)+2)/H)+L.
```

Generalize the operation

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1,2*K,2)).

madhava([]) -> 0;
madhava([H|T]) ->
    L = madhava(T),
    ((( -H rem 4)+2)/H)+L.
```

After generalizing

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1,2*K,2),
              fun(H, L) ->
                  ((( -H rem 4)+2)/H)+L
              end).

madhava([], _Op) -> 0;
madhava([H|T], Op) ->
    L = madhava(T, Op),
    Op(H, L).
```

Generalizing by 0

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1,2*K,2),
              fun(H, L) ->
                  ((( -H rem 4)+2)/H)+L
              end).

madhava([], _Op) -> 0;
madhava([H|T], Op) ->
    L = madhava(T, Op),
    Op(H, L).
```

Generalized initial value

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1,2*K,2),
              fun(H, L) ->
                  ((( -H rem 4)+2)/H)+L
              end, 0).

madhava([], _Op, Z) -> Z;
madhava([H|T], Op, Z) ->
    L = madhava(T, Op, Z),
    Op(H, L).
```

Change the order of the function parameters

```
-module(compute_pi).

-export([by_madhava/1]).
-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1,2*K,2),
              fun(H, L) ->
                  ((( -H rem 4)+2)/H)+L
              end, 0).

madhava([], _Op, Z) -> Z;
madhava([H|T], Op, Z) ->
    L = madhava(T, Op, Z),
    Op(H, L).
```

Parameter reordered

```
-module(compute_pi).

-export([by_madhava/1]).
-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1, 2*K, 2),
        0, fun(H, L) ->
            ((( -H rem 4)+2)/H)+L
        end).

madhava([], Z, _Op) -> Z;
madhava([H | T], Z, Op) ->
    L = madhava(T, Z, Op),
    Op(H, L).
```

Rename function

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*madhava(seq(1, 2*K, 2),
        0, fun(H, L) ->
            ((( -H rem 4)+2)/H)+L
        end).

madhava([], Z, _Op) -> Z;
madhava([H | T], Z, Op) ->
    L = madhava(T, Z, Op),
    Op(H, L).
```

madhava renamed to fold

```
-module(compute_pi).

-export([by_madhava/1]).
-import(lists, [seq/3]).

by_madhava(K) ->
    4*fold(seq(1, 2*K, 2),
        0, fun(H, L) ->
            ((( -H rem 4)+2)/H)+L
        end).

fold([], Z, _Op) -> Z;
fold([H | T], Z, Op) ->
    L = fold(T, Z, Op),
    Op(H, L).
```

Move function

```
-module(compute_pi).

-export([by_madhava/1]).

-import(lists, [seq/3]).

by_madhava(K) ->
    4*fold(seq(1, 2*K, 2),
        0, fun(H, L) ->
            ((( -H rem 4)+2)/H)+L
        end).

fold([], Z, _Op) -> Z;
fold([H | T], Z, Op) ->
    L = fold(T, Z, Op),
    Op(H, L).
```

fold moved to list.erl

```
-module(compute_pi).
-export([by_madhava/1]).
-import(lists, [seq/3]).

by_madhava(K) ->
    4*list:fold(seq(1, 2*K, 2),
        0, fun(H, L) ->
            ((( -H rem 4)+2)/H)+L
        end).
```

```
-module(list).

-export([fold/3]).

fold([], Z, _Op) -> Z;
fold([H | T], Z, Op) ->
    L = fold(T, Z, Op),
    Op(H, L).
```

Move record

```
-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}.

sum_cplx(L)->
    list:fold(L, #cplx{}, fun add/2).
```

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

cplx moved to global.hrl, then extract and inline function

```
-module(cplx).
-export([add/2]).
-include("global.hrl").

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

sum_cplx(L)->
    list:fold(L, #cplx{}, fun add/2).
```

```
-define(START, 1).

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

Extract and inline function

```
-module(cplx).
-export([add/2]).
-include("global.hrl").

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

sum_cplx(L)->
    list:fold(L, #cplx{}, fun add/2).
```

Refactoring data structures

Determine refactoring scope by data flow analysis

- Introduce record
- Upgrade module interface

```
bump(N, {Name, Cnt}) ->
    {Name, Cnt+N}.
pid({Name, _}) ->
    whereis(Name).
```

```
bump(N, R=#inf{cnt=Cnt}) ->
    R#inf{cnt=Cnt+N}.
pid(#inf{name=Name}) ->
    whereis(Name).
```

Refactoring data structures

Determine refactoring scope by data flow analysis

- Introduce record
- Upgrade module interface

```
{match, St, L} =
    regexp:match(S, RE),
strings:substr(S, St, L)
```

```
{match, [{St, L}]} =
    re:run(S, RE),
strings:substr(S, St+1, L)
```

Applications of analysis results

- Call graph visualisation
- Header file splitting based on usage
- "Bad smell" detection

```
con(L) -> con(L, "").
con([], R) -> R;
con([H|T], R) ->
    con(T, R++H).
```

```
stop(S) ->
    gen_server:call(S, stop).
stop_all() ->
    stop(first),
    stop(second).
```

Clustering

- Code restructuring based on component relations
  - Function calls
  - Record and macro usage
- Module clustering
  - Split a large block of modules to more manageable parts
  - Involves splitting of header files
- Function clustering
  - Split a large module into smaller parts
  - Refactoring: move function

Semantic query language

- A language to get information about the Erlang source
- Language concepts:
  - Entities
  - Selectors
  - Properties
  - Filters
- Examples:

```
mods[name=="io"].funs[name==format].refs
@expr.origin
@file.funs.vars[name=="Expl"]
```

- Custom query or predefined query

Semantic query examples

- `@rec.refs.fundef`
- `@fun.calls+`
- ```
  mods[name == "erlang"].
    funs[name == apply and arity == 3].
    refs[type == application].
    sub[index == 3].
    origin[type == atom and value == sum]
  ```
- ```
  @file.funs.vars.bindings.
    origin[type == record_expr].
    reach[type == variable].
    fundef
  ```

Metrics

- Measure the structural complexity of Erlang source code
- Effective Line of code,
  Cohesion of the module,
  Max/min depth of calling,
  McCabe, etc
- Query language
- `show branches_of_recursion for function ({'a','f',1},{'a','g',0}) sum`
- `show max_depth_of_calling for module ('semquery')`

Summary

- RefactorErl: source code analyser and transformer
- Refactoring: helps development
- Analysis: helps maintenance

`http://plc.inf.elte.hu/erlang`