Refactoring and Analysis with RefactorErl

László Lövei

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

June 25, 2009

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

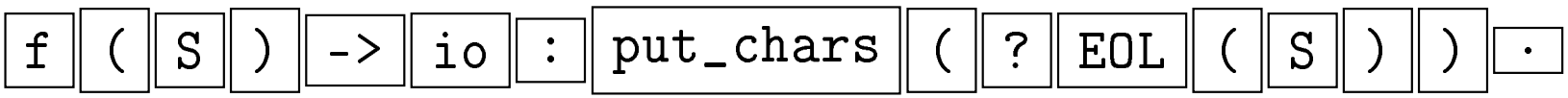
- Work with large code base
- Language coverage
- Code preservation
- Comment preservation
- Layout preservation (indentation)

Three-layered graph model

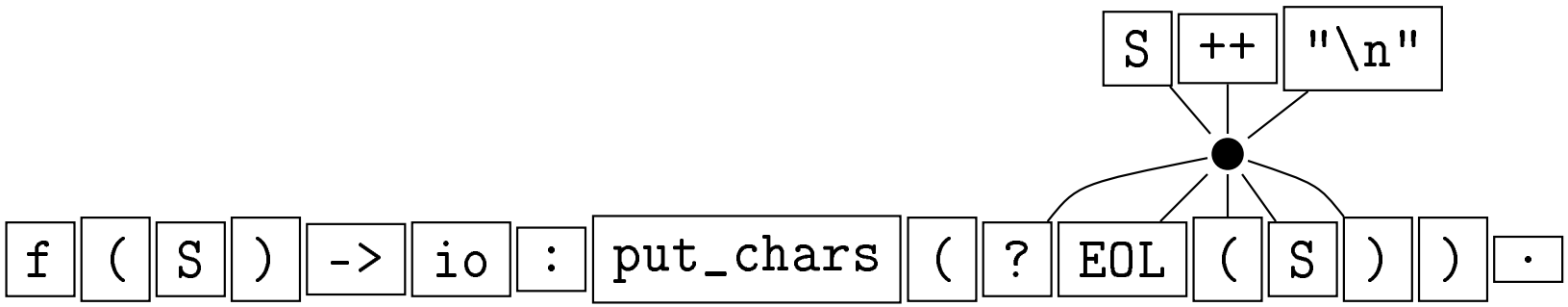
1. Lexical level
   - tokens
   - preprocessing
   - comments, whitespace
2. Syntax level
   - abstract syntax tree
   - files
3. Semantic level
   - module, function, record, variable nodes
   - links to definition and usage

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

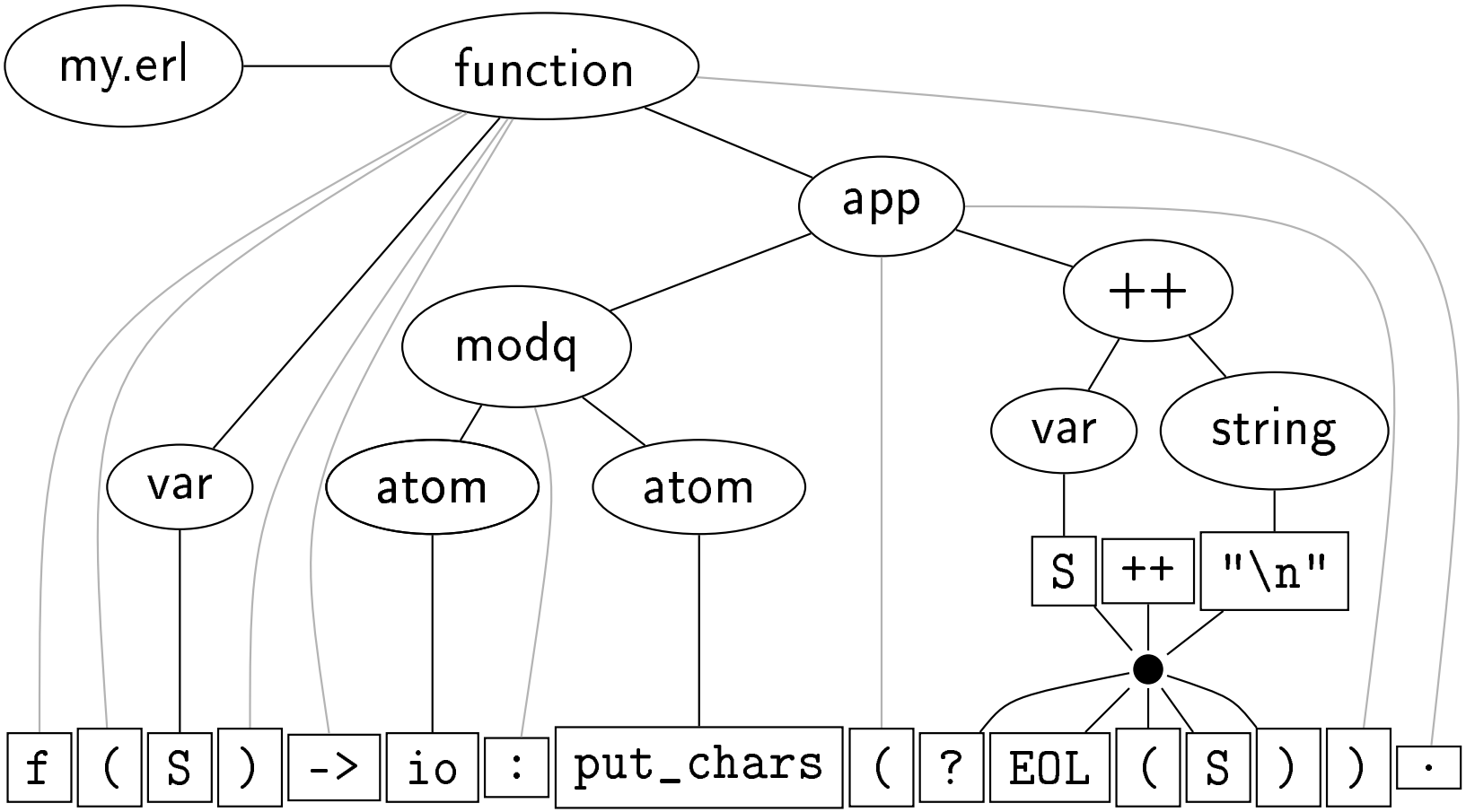
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



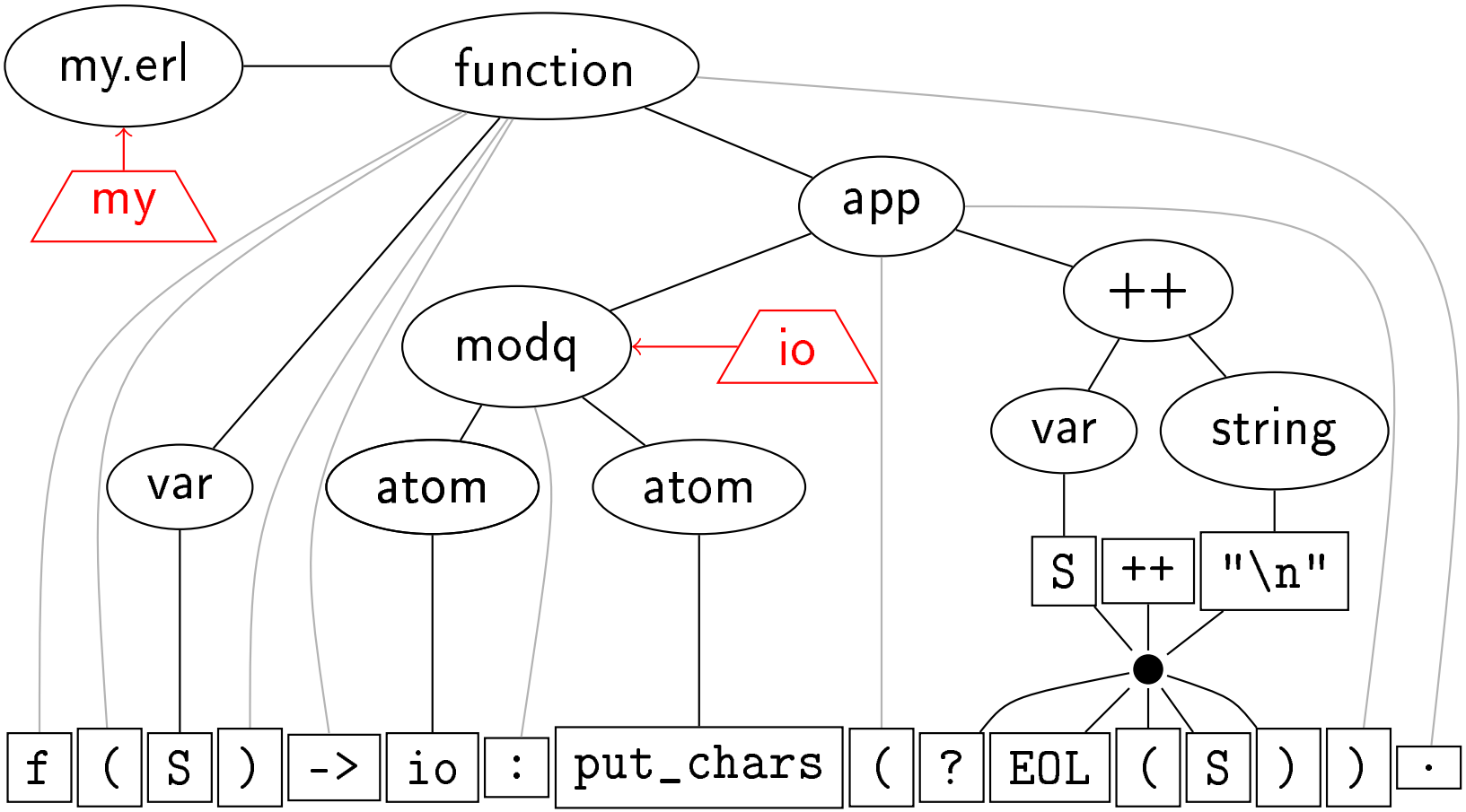
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



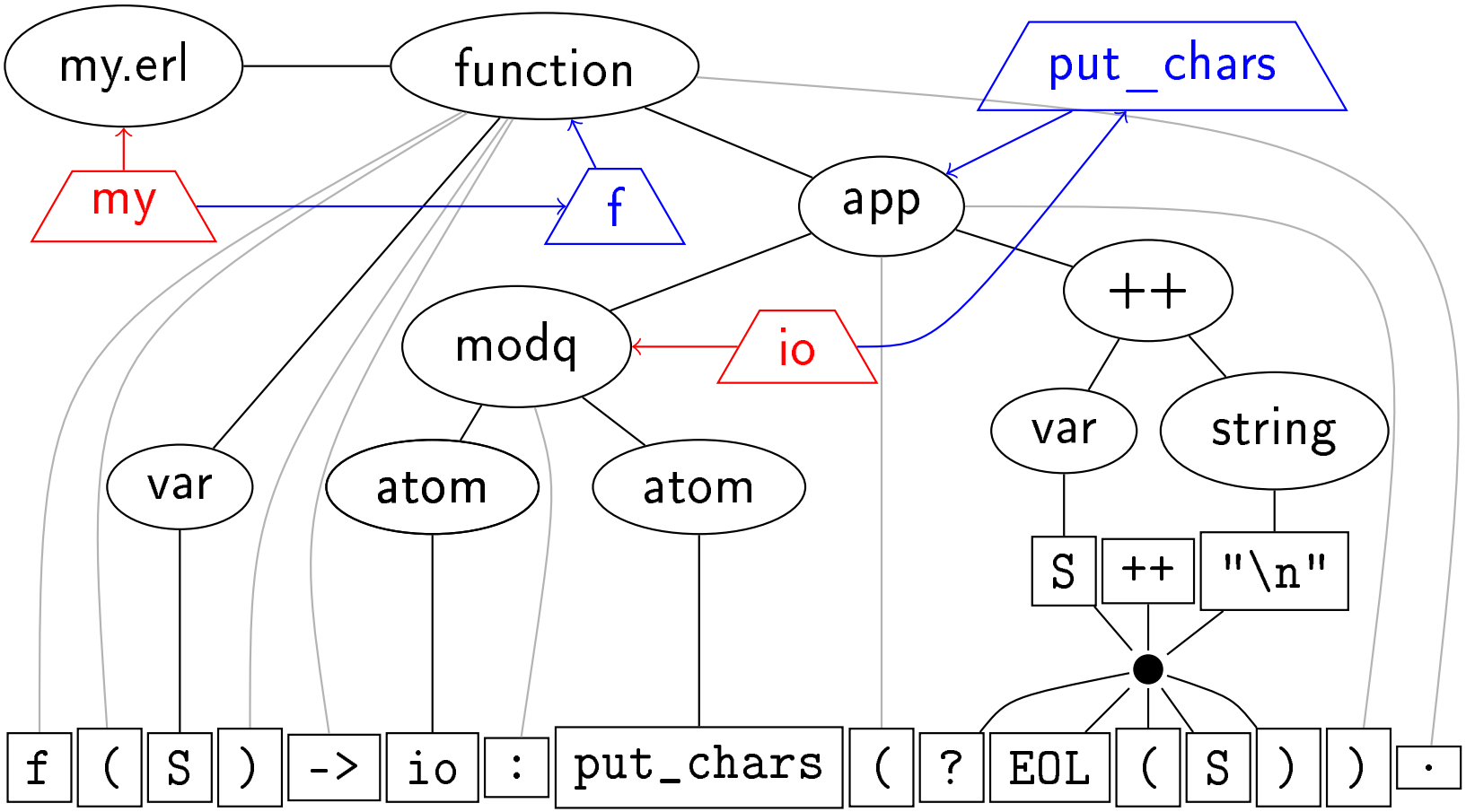
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



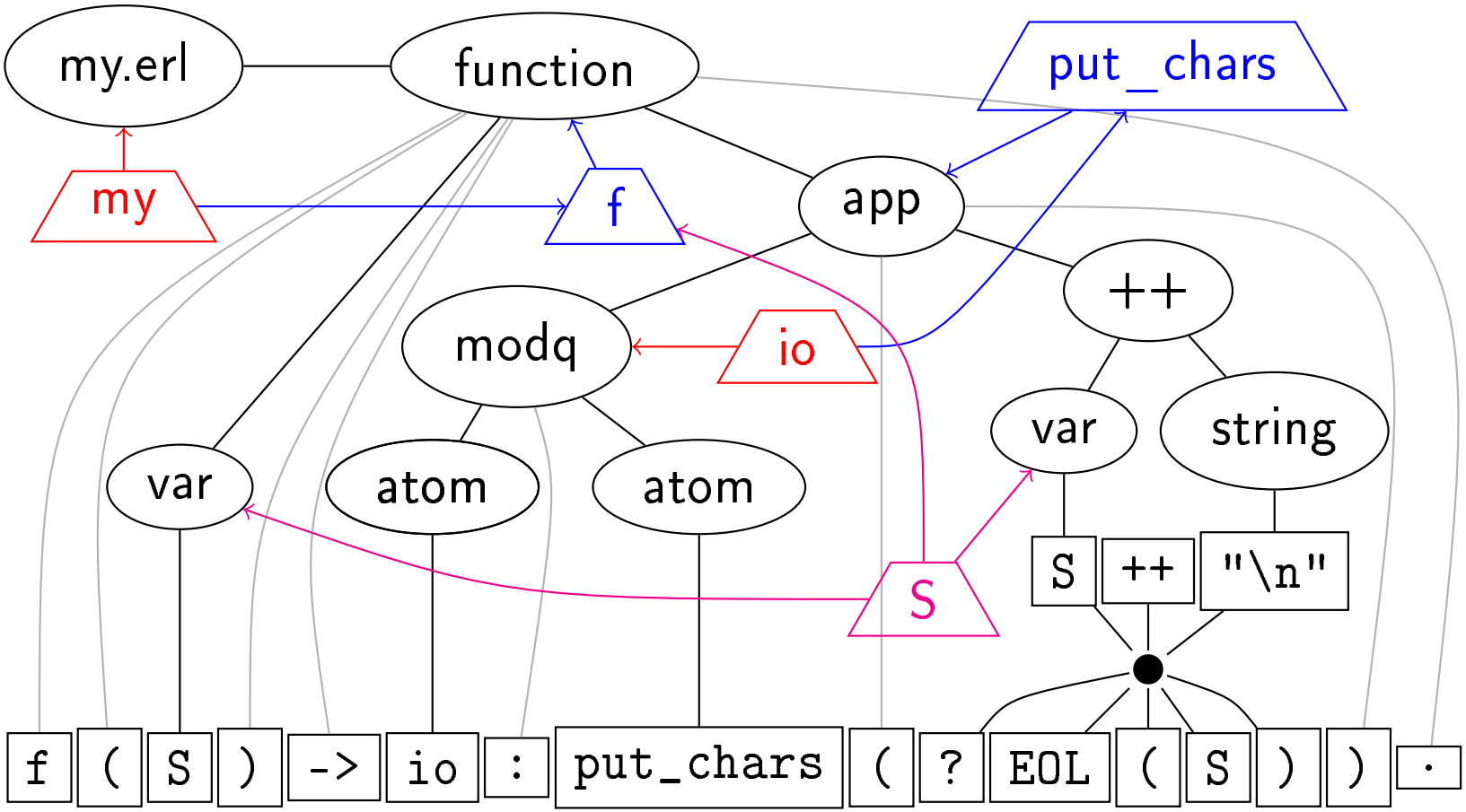
```
-module(my).
-define(EOL(X), X ++ "\n").
f(S) -> io:put_chars(?EOL(S)).
```



```
-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
  - ErlIde, XEmacs, 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

Function interface

- generalize function
- reorder parameters
- tuple parameters

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

Summary

- RefactorErl: source code analyser and transformer
- Helps in development and maintenance

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