DIY Refactorings in Wrangler

Huiqing Li
Simon Thompson

School of Computing
University of Kent

ProTest
property based testing

University of Kent | Computing

Overview

Refactoring.

Wrangler.

DIY Elementary Refactorings.

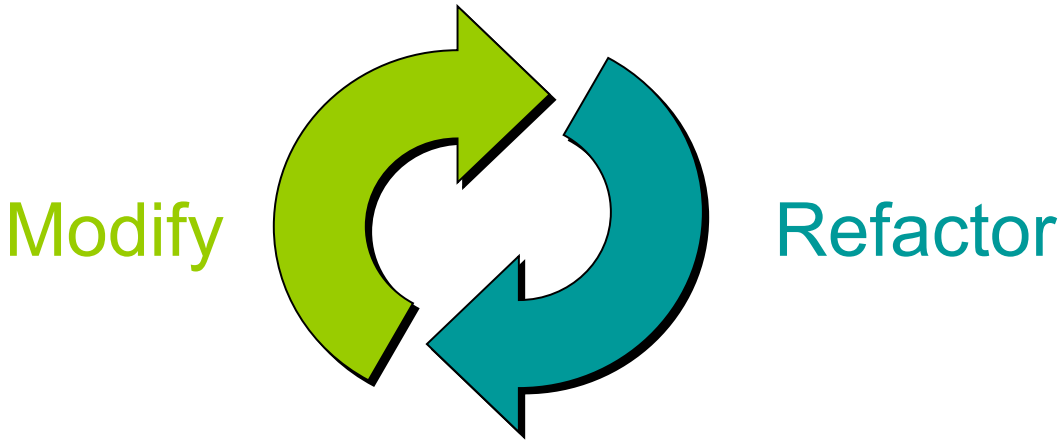
DIY Code Inspections.

DIY Composite Refactorings.

Demo ...

Refactoring

Change how a program works without changing what it does.



Why Refactor?

- Extension and reuse.
  - function extraction, generalisation, ...
- Correct a syntactic decision made earlier.
  - renaming, re-arrange arguments, relocate functions, map to list comprehension, ...
- Counteract decay.
  - clone elimination, module restructure, ...
- API migration.
  - regexp → re, lists:keysearch/2 → list:keyfind/2, ...

How to Refactor?

- By hand … using an editor.
  - flexible, but error-prone.
  - infeasible in the large.
- Tool support.
  - scalable to large codebase.
  - quick and reliable.
  - undo/redo.

Wrangler

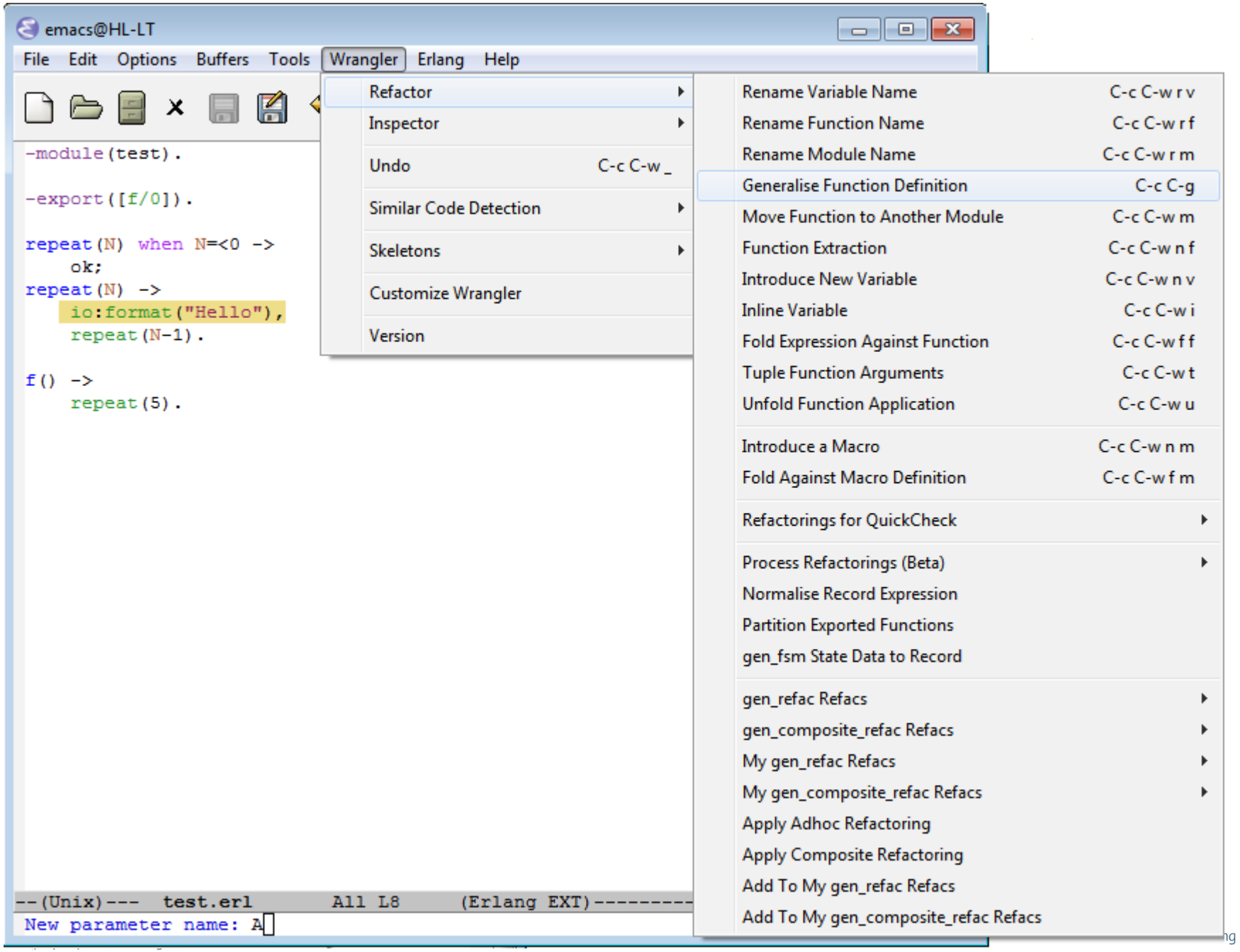
A Refactoring and code smell inspection tool for Erlang

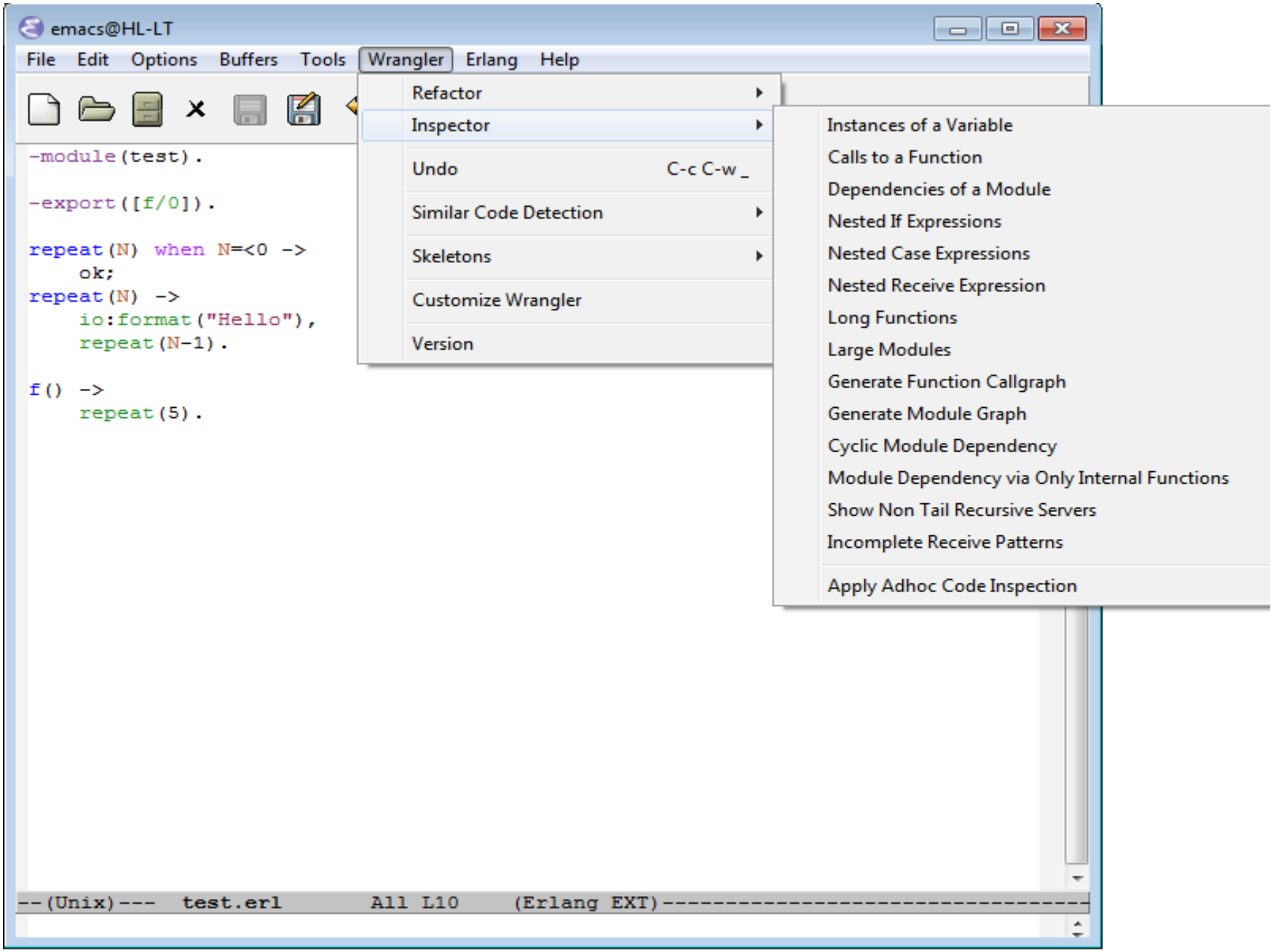
ProTest
property based testing

University of Kent | Computing

Wrangler in a nutshell

- Automate the simple things, and provide decision support tools otherwise.
- Embedded in common IDEs: (X)Emacs, Eclipse.
- Handle full language.
- Faithful to layout and comments.
- Undo
- Build in Erlang, and apply the tool to itself.


emacs@HL-LT
File Edit Options Buffers Tools Wrangler Erlang Help
-module(test).
-export([f/0]).
repeat(N) when N=<0 ->
ok;
repeat(N) ->
io:format("Hello"),
repeat(N-1).
f() ->
repeat(5).
--(Unix)--- test.erl All L8 (Erlang EXT)---
New parameter name: A
Refactor
Inspector
Undo C-c C-w _
Similar Code Detection
Skeletons
Customize Wrangler
Version
Rename Variable Name C-c C-w r v
Rename Function Name C-c C-w r f
Rename Module Name C-c C-w r m
Generalise Function Definition C-c C-g
Move Function to Another Module C-c C-w m
Function Extraction C-c C-w n f
Introduce New Variable C-c C-w n v
Inline Variable C-c C-w i
Fold Expression Against Function C-c C-w f f
Tuple Function Arguments C-c C-w t
Unfold Function Application C-c C-w u
Introduce a Macro C-c C-w n m
Fold Against Macro Definition C-c C-w f m
Refactorings for QuickCheck
Process Refactorings (Beta)
Normalise Record Expression
Partition Exported Functions
gen_fsm State Data to Record
gen_refac Refacs
gen_composite_refac Refacs
My gen_refac Refacs
My gen_composite_refac Refacs
Apply Adhoc Refactoring
Apply Composite Refactoring
Add To My gen_refac Refacs
Add To My gen_composite_refac Refacs


emacs@HL-LT
File Edit Options Buffers Tools Wrangler Erlang Help
Refactor
Inspector
Undo C-c C-w _
Similar Code Detection
Skeletons
Customize Wrangler
Version
Instances of a Variable
Calls to a Function
Dependencies of a Module
Nested If Expressions
Nested Case Expressions
Nested Receive Expression
Long Functions
Large Modules
Generate Function Callgraph
Generate Module Graph
Cyclic Module Dependency
Module Dependency via Only Internal Functions
Show Non Tail Recursive Servers
Incomplete Receive Patterns
Apply Adhoc Code Inspection
-module(test).
-export([f/0]).
repeat(N) when N=<0 ->
ok;
repeat(N) ->
io:format("Hello"),
repeat(N-1).
f() ->
repeat(5).
--(Unix)--- test.erl All L10 (Erlang EXT)----------

Wrangler

Code
clone detection
and removal

Module
structural
improvement

Basic Refactorings and Code inspections

Demo

Wrangler

So what are the limitations?

- Only a set of `core' refactorings supported.
- Only elementary refactorings are supported, i.e., batch refactorings are not supported.
- Wrangler is designed as a black-box.

Wrangler

Code Clone Detection and Removal	Module structural improvement	Template- and rule- base API for DIY basic refactorings/ code inspections	A DSL for DIY composite refactorings

Basic Refactorings and Code inspections

Wrangler

Wrangler built-in refactorings + Refactorings contributed by users + User's own refactorings

DIY Basic Refactorings

Design criteria

- We assume you can program Erlang …

  … but don't want to learn the internal syntax or details of our representation and libraries.

- We aim for simplicity and clarity.

DIY elementary refactorings

Context available for condition analysis

Traversals describe how rules are applied

Erlang **Behaviour** for refactoring

Rules describe transformations

Templates describe code fragments

Templates

- Templates are enclosed in the ?T macro call.
- Meta-variables in templates are Erlang variables end in @, e.g. F@, Args@@, Guards@@@.
- Meta-atoms in templates are Erlang atoms end in @, e.g. f@.

Templates

- Examples

?T(''F@(1, 2)'') F@ matches a single element.

?T(''spawn(Args@@)'') Args@@ matches a sequence of elements of some kind.

?T("f@(Args@@)when Guard@@-> Body@@;") matches a function clause.

?T("f@(Args@@)when Guard@@-> Body@@.") matches a function definition of a single function clause.

Rules

?RULE(Template, NewCode, Cond)

The old code, the new code and the pre-condition.

```
rule({M,F,A}, N) ->
    ?RULE(?T("F@(Args@@)"),
              begin
                 NewArgs@@=delete(N, Args@@),
                 ?TO_AST("F@(NewArgs@@)")
              end,
              refac_api:fun_define_info(F@) == {M,F,A}).

delete(N, List) ->  … delete Nth elem of List …
```

Information in the AAST

Wrangler uses the syntax_tools AST, augmented with information about the program semantics.

API functions provide access to this.

Variables bound, free and visible at a node.

Location information.

All bindings (if a vbl).

Where defined (if a fn).

Atom usage info: name, function, module etc.

Process info …

Collecting Information

?COLLECT(Template, Collector, Cond)

- The template to match.
- In information to extract (“collect”).
- Condition on when to collect information.

Collecting information

```
?COLLECT(?T("f@(Pars@@) when G@@ -> B@@;"),
                lists:nth(Nth, Pars@@),
                refac_api:fun_def_info(F@) =={M, F, A})
```

Collect the nth parameters

```
?COLLECT(?T("Body@@, V@=Expr@, V@")"),
                {_File@,  refac_api:start_end_loc(_This@)},
                 refac_api:type(V@) == variable andalso
                 [ _ ]  == refac_api:refs(V@))
```

_File@ current file _This@ subtree matching ?T(...)

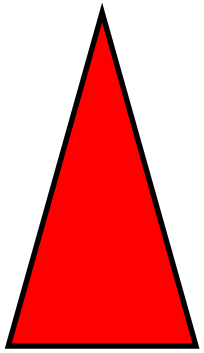
Unnecessary match

ProTest property based testing

University of Kent Computing

AST Traversals

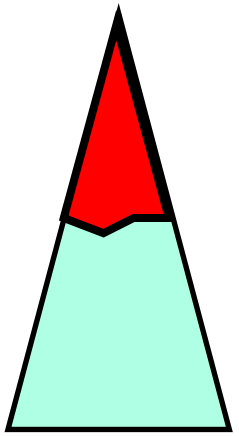
?FULL_TD_TP(Rules, Scope)



- Traverse top-down.
- At each node, apply first of Rules to succeed …
- TP = “Type Preserving”.

AST Traversals

?STOP_TD_TU(Collectors, Scope)



- Traverse top-down.
- … apply all of the collectors to succeed…
- TU = “Type Unifying”.

Generic Refactoring Behaviour

Behaviour gen_refac encapsulates what a refactoring needs to provide

Callback functions:

- input_par_prompts/0: prompts for interactive input.
- select_focus/1: what to do with focus information.
- selective/0: selective refactoring or not.
- pre_cond_check/1: check pre-conditions.
- transform/1: if the pre-condition is ok, do the transformation.

DIY Refactorings In Wrangler

```
-module(refac_replace_append). %%module name is also refactoring name.

-include_lib("wrangler/lib/wrangler.hrl")

-behaviour(gen_refac).

-export([input_par_prompts/0, select_focus/1,check_pre_cond/1,
         selective/0,transform/1]). %% Callback functions.

input_par_prompts() -> []. %% No user input is needed.

select_focus(_Args) -> {ok, none}. %% No focus selection is need.

check_pre_cond(_Args) -> ok.  %% No pre-condition.

selective() -> true. %% Allow selective refactoring.

transform(_Args=#args{search_paths=SearchPaths})->
  ?FULL_TD_TP([rule_replace_append()], SearchPaths).

rule_replace_append() ->
    ?RULE(?T("F@(L1@, L2@)"), ?TO_AST("L1@++L2@"),
          {lists,append,2} == refac_api:fun_def_info(F@)).
```

DIY Refactorings In Wrangler

```
rule1({M,F,A}, I, J) ->  %% transform the function definition itself.
    ?RULE(?T("f@(Args@@) when Guard@@ -> Bs@@;"),
              begin
                 NewArgs@@=swap(Args@@,I,J),
                 ?TO_AST("f@(NewArgs@@) when Guard@@->Bs@@;")
              end,
              api_refac:fun_define_info(f@) == {M, F, A}).

rule2({M,F,A}, I, J) ->  %% Transform the different kinds of function applications.
     ?RULE(?FUN_APPLY(M,F,A),
               begin
                  Args=api_refac:get_app_args(_This@),
                  NewArgs=swap(Args, I, J),
                  api_refac:update_app_args(_This@,NewArgs)
               end, true).

rule3({_M, F, A}, I, J)  ->   %% transform the type spec.
   ?RULE(?T("Spec@"),  api_spec:swap_arg_types_in_spec(_This@, I, J),
             api_spec:is_type_spec(Spec@, {F, A})).
```

Transformation rules for swapping arguments

DIY Code Inspections In Wrangler

```
calls_to_specific_function(input_par_prompts) ->
    ["Module name: ", "Function name: ", "Arity: "];

calls_to_specific_function(_Args=#args{user_inputs=[M,F,A],
                                       search_paths=SearchPaths}) ->
    {M1, F1, A1}={list_to_atom(M), list_to_atom(F), list_to_integer(A)},
    ?FULL_TD_TU([?COLLECT_LOC(?FUN_APPLY(M1, F1, A1), true)],
                [SearchPaths]).
```

Collect falls to a specific Function

Demo

DIY Composite Refactorings

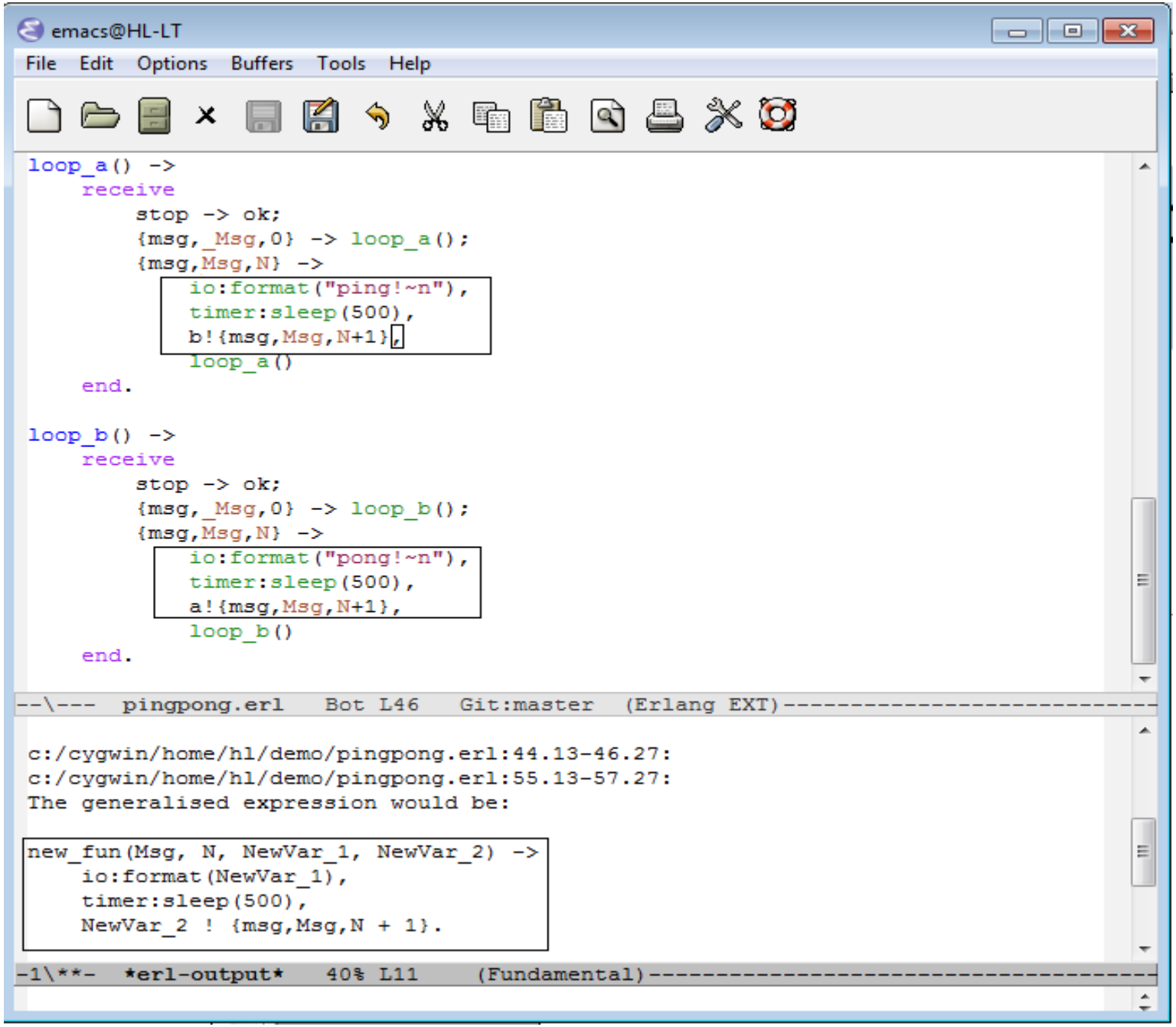
Composite Refactoring

A set of elementary refactorings to be applied in a sequence in order to achieve a complex refactoring effect.

DIY Composite Refactorings

Example 1.

- Batch renaming of function names from CamelCase to camel_case.
  - Rename, Rename, Rename, ...


emacs@HL-LT
File Edit Options Buffers Tools Help
loop_a() ->
receive
stop -> ok;
{msg,_Msg,0} -> loop_a();
{msg,Msg,N} ->
io:format("ping!~n"),
timer:sleep(500),
b!{msg,Msg,N+1},
loop_a()
end.
loop_b() ->
receive
stop -> ok;
{msg,_Msg,0} -> loop_b();
{msg,Msg,N} ->
io:format("pong!~n"),
timer:sleep(500),
a!{msg,Msg,N+1},
loop_b()
end.
--\--- pingpong.erl Bot L46 Git:master (Erlang EXT)
c:/cygwin/home/hl/demo/pingpong.erl:44.13-46.27:
c:/cygwin/home/hl/demo/pingpong.erl:55.13-57.27:
The generalised expression would be:
new_fun(Msg, N, NewVar_1, NewVar_2) ->
io:format(NewVar_1),
timer:sleep(500),
NewVar_2 ! {msg,Msg,N + 1}.
-1**- *erl-output* 40% L11 (Fundamental)

DIY Composite Refactorings

Example 2.

- Clone elimination.
  - generate new function,
  - rename function,
  - rename variables,
  - re-order parameters,
  - add to export,
  - folding, folding, ...

DIY Composite Refactorings

Issues to handle

- Generation of refactoring commands.
- Tracking of program entity names.
- Handling of failure.
- User control over the execution of refactorings.

DIY Composite Refactorings

Generation of refactoring cmds

- Manual vs. automatic.
- Static vs. dynamic.

Generation of Refactoring Cmds

```
-spec rename_fun(File::filename(), FunNameArity::{atom(), integer()},
                 NewName::atom()) -> ok | {error, Reason::string()}.
```

(a) type spec of the 'rename function' refactoring.

```
-spec rename_fun(File::filename() | fun((filename()) -> boolean()),
                 FunNameArity::{atom(), integer()}
                               | fun(({atom(),integer()}) -> boolean()),
                 NewName::atom()
                   |{generator, fun(({filename(), {atom(), integer()}})
                                    -> atom())}
                   |{user_input,fun(({filename(), {atom(), integer()}})
                                   -> string())},
                 Lazy :: boolean())
          -> [{refactoring, rename_fun, Args::[term()]}] |
             {{refactoring, rename_fun, Args::[term()]}, function()}.
```

(b) type spec of the 'rename function' command generator.

Generation of Refactoring Commands

?refac_(CmdName, Args, Scope)

```
?refac_(rename_fun,
        [{file, fun(_File)-> true end},
         fun({F, _A}) ->
              camelCase_to_camel_case(F) /= F
         end,
         {generator, fun({_File, F,_A}) ->
                              camelCase_to_camel_case(F)
                     end}],
        SearchPaths).
```

Example: Generation of refactoring cmds that rename function names in camelCase to camel_case.

Track Program Entity Names

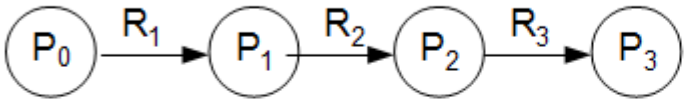
- The name of an entity (function, module, process name) referred by a refactoring may have been changed by one or more previous refactorings.
- Manual tracking of names infeasible.
- Wrangler tracks the renaming history in the background …
- … uses use the macro ?current to retrieve the latest name of an entity.

Handling of Failure

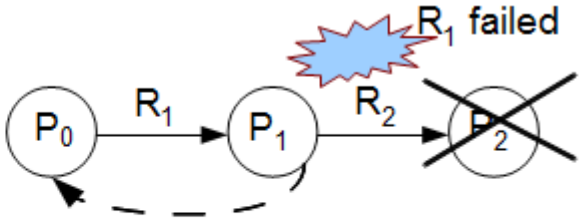
What to do if a refactoring fails?

- Atomic composite refactoring.

  ?atomic(Refacs)



a) A successful execution of ?atomic([R1,R2,R3]).



b) A failed execution of ?atomic([R1,R2,R3]).

Handling of Failure

What to do if a refactoring fails?

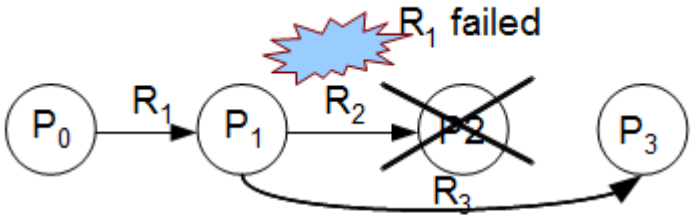
```
?atomic(
    ?refac_(rename_fun,
            [{file, fun(_File)-> true end},
             fun({F, _A}) ->
                  camelCase_to_camel_case(F) /= F
             end,
             {generator, fun({_File, F,_A}) ->
                                camelCase_to_camel_case(F)
                            end}],
            SearchPaths))
```

Handling of Failure

What to do if a refactoring fails?

- Non-atomic composite refactoring.

  ?non_atomic(Refacs)



c) An execution of ?non_atomic([R1,R2,R3]).

Handling of Failure

What to do if a refactoring fails?

```
?non_atomic(
    ?refac_(rename_fun,
            [{file, fun(_File)-> true end},
             fun({F, _A}) ->
                  camelCase_to_camel_case(F) /= F
             end,
             {generator, fun({_File, F,_A}) ->
                                camelCase_to_camel_case(F)
                            end}],
            SearchPaths))
```

User control

- Allow the user to control whether a refactoring should be performed.

?interative(Refacs).

```
?interactive(
  ?non_atomic(
     ?refac_(rename_fun,
                [{file, fun(_File)-> true end},
                  fun({F, _A}) ->
                        camelCase_to_camel_case(F) /= F
                  end,
                  {generator, fun({_File, F,_A}) ->
                                      camelCase_to_camel_case(F)
                                end}],
                SearchPaths)))
```

A DSL for Compound Refactorings

$RefacName ::= rename\_fun \mid rename\_mod \mid rename\_var \mid new\_fun \mid gen\_fun \mid ...$

$PR ::= \{refactoring,\ RefacName,\ Args\}$

$CR ::= PR$

$\quad \mid \{interactive,\ Qualifier,\ [PRs]\}$

$\quad \mid \{repeat\_interactive,\ Qualifier,\ [PRs]\}$ *%% repetitive interaction.*

$\quad \mid \{if\_then,\ Cond,\ CR\}$ *%% conditional cmd generation.*

$\quad \mid \{while,\ Cond,\ Qualifier,\ CR\}$ *%% repetitive conditional cmd generation.*

$\quad \mid \{Qualifier,\ [\ CRs\ ]\}$

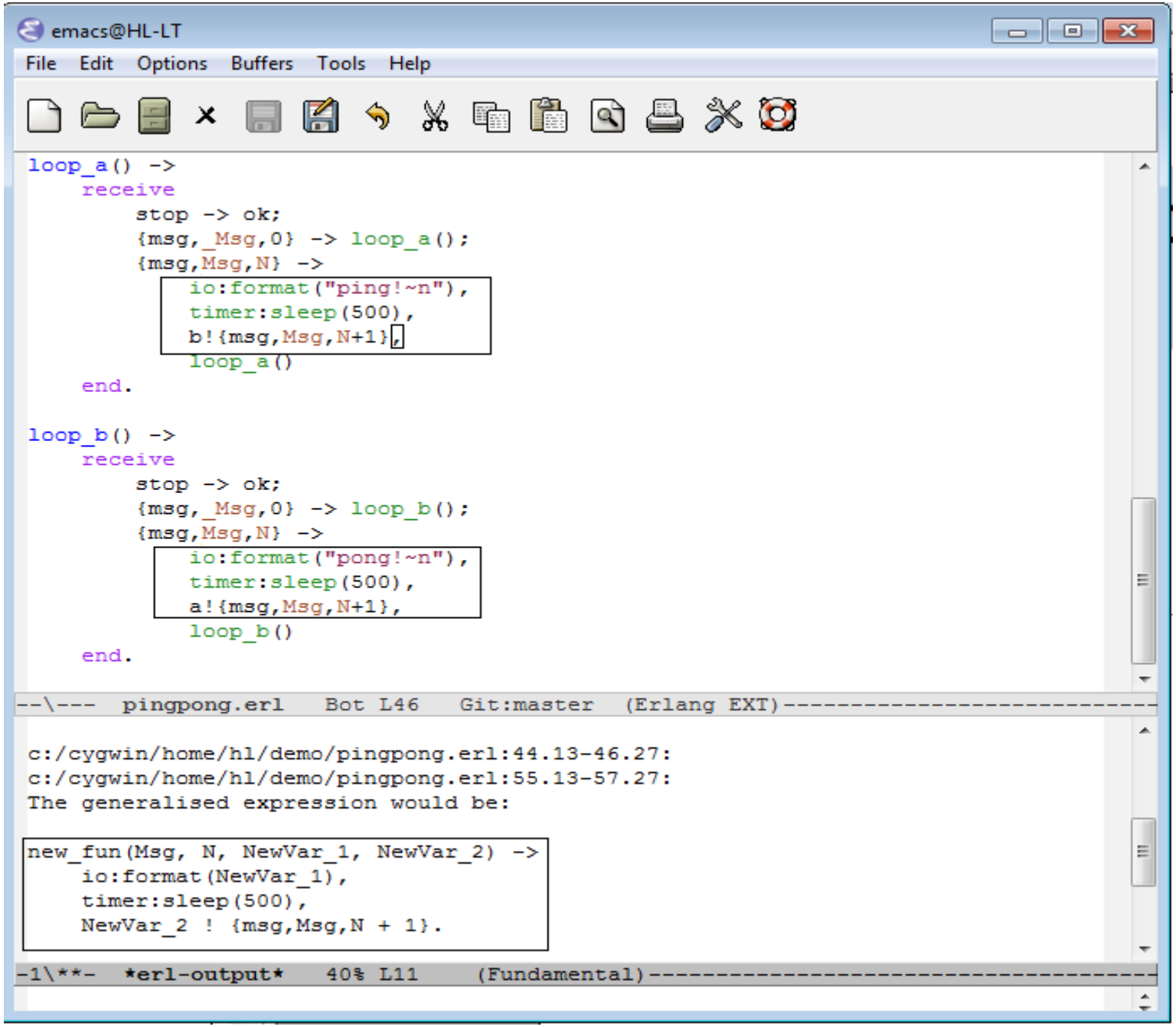
$PRs ::= PR \mid PRs,\ PR$

$CRs ::= CR \mid CRs,\ CR$

$Qualifier ::= atomic \mid non\_atomic$

$Args ::= ...A\ list\ of\ Erlang\ terms...$

$Cond ::= ...An\ Erlang\ expression\ that\ evaluates\ to\ a$ `boolean` $value...$


emacs@HL-LT
File Edit Options Buffers Tools Help
loop_a() ->
receive
stop -> ok;
{msg,_Msg,0} -> loop_a();
{msg,Msg,N} ->
io:format("ping!~n"),
timer:sleep(500),
b!{msg,Msg,N+1},
loop_a()
end.
loop_b() ->
receive
stop -> ok;
{msg,_Msg,0} -> loop_b();
{msg,Msg,N} ->
io:format("pong!~n"),
timer:sleep(500),
a!{msg,Msg,N+1},
loop_b()
end.
--\--- pingpong.erl Bot L46 Git:master (Erlang EXT)
c:/cygwin/home/hl/demo/pingpong.erl:44.13-46.27:
c:/cygwin/home/hl/demo/pingpong.erl:55.13-57.27:
The generalised expression would be:
new_fun(Msg, N, NewVar_1, NewVar_2) ->
io:format(NewVar_1),
timer:sleep(500),
NewVar_2 ! {msg,Msg,N + 1}.
-1**- *erl-output* 40% L11 (Fundamental)

DIY Composite Refactorings

Example 2.

- Clone elimination.
  - generate new function,
  - rename function,
  - rename variables,
  - re-order parameters,
  - add to export,
  - folding, folding, folding, …

Demo

Generic Refactoring Behaviour

Behaviour gen_composite_refac encapsulates what a composite refactoring needs to provide.

Callback functions:

- input_par_prompts/0: prompts for interactive input.
- select_focus/1: what to do with focus information.
- composite_refac/1: defines the refactoring script.

Find out more

Latest release of Wrangler: 1.0

www.cs.kent.ac.uk/projects/wrangler

Papers:

A User-extensible Refactoring Tool for Erlang Programs. Huiqing Li and Simon Thompson. 2011. http://www.cs.kent.ac.uk/pubs/2011/3171/index.html

A Domain-Specific Language for Scripting Refactorings in Erlang. Huiqing Li and Simon Thompson. 2011. http://www.cs.kent.ac.uk/pubs/2011/3172/index.html

Installation: Mac OS X and Linux

Download Wrangler-1.0 from

http://www.cs.kent.ac.uk/projects/wrangler/

or get it from

https://github.com/RefactoringTools/wrangler

In the wrangler directory

```
./configure
make
(sudo) make install
```

Installation: Mac OS X and Linux

Add to `~/.emacs` file:

```
(add-to-list 'load-path
    "/usr/local/lib/erlang/lib/wrangler-<VSN>/elisp")
(require 'wrangler)
```

If you're installing emacs now, then you add the following lines to your `~/.emacs` file

```
(setq load-path (cons "/usr/local/otp/lib/tools-<ToolsVsn>/emacs"
                      load-path))
(setq erlang-root-dir "/usr/local/otp")
(setq exec-path (cons "/usr/local/otp/bin" exec-path))
(require 'erlang-start)
```

Installation: Windows

Requires R11B-5 or later + Emacs

Download installer from

http://www.cs.kent.ac.uk/projects/wrangler/

Requires no other actions.

Installation: Eclipse + ErlIDE

Requires Erlang R11B-5 or later, if it isn't already present on your system.

On Windows systems, use a path with no spaces in it.

Install Eclipse 3.5, if you didn't already.

All the details at

http://erlide.sourceforge.net/

Starting Wrangler in Emacs

Open emacs, and open a `.erl` file.

`M-x erlang-refactor-on` or ...

... `C-c, C-r`

New menu: Wrangler

Customise for dir

Undo `C-c, C-w, _`

Preview Feature

Preview changes before confirming the change

Emacs `ediff` is used.

Stopping Wrangler in Emacs

`M-x erlang-refactor-off` to stop Wrangler

Shortcut `C-c, C-r`

Carrying on …

Try on your own project code …

Feedback:

erlang-refactor@kent.ac.uk or

H.Li@kent.ac.uk