Let’s Make Refactoring Tools User-extensible!

Simon Thompson and Huiqing Li
School of Computing
University of Kent
Refactoring

Change how a program works without changing what it does
Why refactor?

Extension and reuse

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.
Why refactor?

Extension and reuse

```
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_a() ->
    receive
        stop -> ok;
        {msg, _Msg, 0} -> loop_a();
        {msg, Msg, N} ->
            body(Msg, N),
            loop_a()
    end.

body(Msg, N) ->
    io:format("ping!~n"),
    timer:sleep(500),
    b ! {msg, Msg, N - 1}.
```
Why refactor?

Counteract decay ... comprehension

“Clones considered harmful”: detect and eliminate duplicate code.

Improve the module structure: remove loops, for example.
How to refactor?

By hand … using an editor.

Flexible … but error-prone.
Infeasible in the large.

Tool supported.

Handle atoms, names, side-effects, …
Scalable to large-code bases.
Integrated with tests, macros, …
Wrangler

Clone detection and removal

Module structure improvement

Basic refactorings: structural, macro, process and test-framework related
Wrangler in a nutshell

Automate the simple things, and …
… provide decision support tools otherwise.
Embed in common IDEs: emacs, eclipse, …
Handle full language, multiple modules, tests, …
Faithful to layout and comments.
Build in Erlang and apply the tool to itself.
-module(test_camel_case).
-export([thisIsAFunction/2, this_is_a_function/2, thisIsAnotherFunction/2]).

thisIsAFunction(X, Y) ->
   this_is_a_function(X, Y).

this_is_a_function(X, Y) ->
   thisIsAnotherFunction(X, Y).

thisIsAnotherFunction(X, Y) ->
   X+Y.

---

test_camel_case.erl  All (13,0)  (Erlang EXT Flymake)
Wrangler started.
-module(test_camel_case).
-export([thisIsAFunction/2, this_is_a_function/2, thisIsAnotherFunction/2]).

thisIsAFunction(X, Y) ->
    this_is_a_function(X, Y).

this_is_a_function(X, Y) ->
    thisIsAnotherFunction(X, Y).

thisIsAnotherFunction(X, Y) ->
    X+Y.

---

Wrangler started.
I'd like to swap argument positions, or delete an argument, or upgrade to a new API, …
## Two extensions

<table>
<thead>
<tr>
<th>API</th>
<th>DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describe entirely new ‘atomic’ refactorings from scratch.</td>
<td>A language to script composite refactorings on top of simpler ones.</td>
</tr>
<tr>
<td>e.g. swap args, delete argument.</td>
<td>e.g. remove clone, migrate API.</td>
</tr>
</tbody>
</table>
API
API 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 …

… rather than complete coverage.
Integration

Describe refactorings by a behaviour ...

... that's Erlang-speak for a set of callbacks.

Integration with emacs for execution ...

... which gives preview, undo, interactive behaviour etc. “for free”.
Generalisation

Describe expressions in Erlang ...

```erlang
loop_a() ->
    receive
        stop -> ok;
        {msg, _Msg, 0} -> loop_a();
        {msg, Msg, N} ->
            body(Msg, N),
            loop_a()
    end.

body(Msg, N) ->
    io:format("ping!~n"),
    timer:sleep(500),
    b ! {msg, Msg, N - 1}.
```

Generalisation

... how expressions are transformed ...

```
loop_a() ->
    receive
        stop -> ok;
        {msg, _Msg, 0} -> loop_a();
        {msg, Msg, N} ->
            body(Msg, N).
        loop_a()
    end.

body(Msg, N) ->
    io:format("ping!~n"),
    timer:sleep(500),
    b ! {msg, Msg, N - 1}.
```
Generalisation

...and its context and scope.
Generalisation

Pre-conditions for refactorings

Can’t generalise over an expression that contains free variables ...

… or use the same name as an existing variable for the new variable.
Wrangler API

- **Context**: available for pre-conditions
- **Traversals**: describe how rules are applied
- **Rules**: describe transformations
- **Templates**: describe expressions
Templates

Templates are enclosed in the \texttt{?T} macro call.

Meta-variables in templates are Erlang variables ending in \texttt{@}, e.g. \texttt{F@, Arg@@, Guards@@@}. 

\begin{align*}
\text{Tests:} & \quad \text{T("M:F@(1,2)")} \quad \text{F@ matches a single element.} \\
\text{Tests:} & \quad \text{T("spawn(Args@@)"')} \quad \text{Args@@ matches a sequence of elements of some kind.} \\
\text{Tests:} & \quad \text{T("spawn(Arg1@, Arg2@, Args@@")')} 
\end{align*}
Apply function $f$: $f(1,a,3)$

```prolog
fun test_swap_args:f/3(1,a,3).
fun f/3(1,a,3).
test_swap_args:f(1,a,3).
apply(test_swap_args, f, [1,a,3]).
spawn(test_swap_args, f, [1,a,3]).
As = [1,a,3], apply(test_swap_args, f, As).
apply(fun test_swap_args:f/3, [1,2,3]).
```

Different concrete syntax for application.

Replace with single

```prolog
?FUN_APPLY(M, F, A)
```
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.
• The information to extract (“collect”).
• Condition on when to collect the information.

?COLLECT(?T("Body@@, V@=Expr@, V@"),
   {_File@, refac_api:start_end_loc(_This@)},
   refac_api:type(V@) == variable).

_File@  current file   _This@  subtree matching ?T(…)
Traversals

**?STOP_TD_TU(Collectors, Scope)**
- Traverse top-down
- … apply all of the Collectors to succeed …
- … only visit sub-nodes if no collector has fired.
- **TU** = “Type unifying”.

**?FULL_TD_TP(Rules, Scope)**
- Traverse top-down
- At each node, apply first of Rules to succeed …
- **TP** = “Type preserving”.
DSL
How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Much of what we know about how programmers refactor in the wild is based on studies that examine just a few software projects. Researchers have rarely taken the time to replicate these studies in other contexts or to examine the assumptions on which they are based. To help put refactoring research on a sound scientific basis, we draw conclusions using four data sets spanning more than 13,000 developers, 240,000 tool-assisted refactorings, 2,500 developer hours, and 3,400 version control commits. Using these data, we cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, we find that programmers frequently do not indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, we were able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes. By confirming assumptions and replicating studies made by other researchers, we can have greater confidence that those researchers’ conclusions are generalizable.

Up to 90% of refactorings done by hand

Some 40% of refactorings performed using tools are done in batches.

a single research method: Weißgerber and Diehl’s study of 3 open source projects [18]. Their research method was to

determine whether on every day on which refactoring took place, non-refactoring code changes also took place. What we can learn from this depends on the relative frequency of high-level and mid-to-low-level refactorings. If not have explored. In this paper we use both of these methods to confirm — and cast doubt on — several conclusions that have been published in the refactoring literature.
Composite refactorings

A sequence of simpler refactorings which together achieve a complex effect.

Example: transform all camelCase identifiers within a project into camel_case.
Rename function  
Rename variables  
Reorder variables  
Add to export list  
Fold* against the def. 

new_fun(Msg, N, NewVar_1, NewVar_2) ->  
  io:format(NewVar_1),  
  timer:sleep(500),  
  NewVar_2 ! {msg,Msg,N + 1}.  

--- 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:
Not just a script ...

Tracking changing names and positions.
Generating refactoring commands.
Dealing with failure.
User control of execution.

... we're dealing with the **pragmatics** of composition, rather than just the theory.
Generators

Refactoring functions modified to take descriptions of arguments, rather than concrete arguments.

```
rename_fun(Module, {Fun, Arity}, NewName) -> ok | error

rename_fun(fun(Module) -> boolean(),
             fun({Fun, Arity}) -> boolean(),
             fun(Module, {Fun, Arity}) -> atom(),
             boolean()) ->
    { [ {refactoring, rename_fun, Args} ], fun }
```
Generation: camel case

?refac_(CmdName, Args, Scope)

Args: modules, camelCase functions, new names.

?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).
Automation

Don't have to describe each command explicitly: allow *conditions* and *generators*.

Allow lazy generation … return a refactoring command together with a *continuation*.

*Track names*, so that ?current(foo) gives the ‘current’ name of an entity foo at any point in the refactoring.
Handling failure

Transaction: if one part fails, abandon the whole.

Otherwise: continue even when failure.
Handling interaction

Interactive: choose the cases to perform.

Non-interactive: perform all cases.
Handling repetition

Condition: *repeat* while a condition is true.

Interactively: *ask* whether to repeat.
Building a DSL

Domain specific language to support options of atomicity, interactivity etc.

*Embed* in Erlang to leverage the language e.g. to define conditions and generators.

Use Erlang to represent the language, and *macros* to support.
- Rename function
- Rename variables
- Reorder variables
- Add to export list
- Fold* against the def.

new_fun(Msg, N, NewVar_1, NewVar_2) ->
    io:format(NewVar_1),
    timer:sleep(500),
    NewVar_2 ! {msg, Msg, N + 1}.
Clone removal: top level

Atomic as a whole … non-atomic components OK.

Not just an API: ?atomic etc. modify interpretation of what they enclose …

?atomic([?interactive( RENAME FUNCTION )
             ?refac_( RENAME ALL VARIABLES OF THE FORM NewVar* )
             ?repeat_interactive( SWAP ARGUMENTS )
             ?if_then( EXPORT IF NOT ALREADY )
             ?non_atomic( FOLD INSTANCES OF THE CLONE )
]):
Erlang and DSL

?refac_(rename_var,
[M,
 begin
  {_, F1, A1} = ?current(M,F,A),
  {F1, A1}
 end,
 fun(X) ->
   re:run(atom_to_list(X), "NewVar*")/=nomatch
 end,
 {user_input, fun({_, _, V}) ->
   lists:flatten(io_lib:format
       "Rename variable ~p to: ", [V]))
 end},
 SearchPaths)
Remove bug
preconditions
Remove bug preconditions

Scenario: building Erlang models for C code.

For buggy code, want to avoid hitting the same bugs all the time

Add bug precondition macros …

… but want to remove in delivered code.

Step 1: simple rules

replace_bug_cond_macro_rule() ->
  ?RULE(?T("Expr@"),
       ?TO_AST("false"),
       is_bug_cond_macro(Expr@)).

logic_rule_1() ->
  ?RULE(?T("not false"),?TO_AST("true"),true).
Step 2: tidy up

case false of
  true  -> com_cfg:initial_value(Sig);
  false -> get_shadow_value(Id, S)
end.

Simplifies to  get_shadow_value(Id, S).
Step 3: inline variables

route_data_next(S,_, [[SrcKind,SrcId], Dst, Val]], _) ->
  % clear gateway pending flag
  NewS = set_gateway_pending(S, SrcKind, SrcId, false),
  S2   = NewS,
  copy_to_destination(S2, Dst, Val).
API Migration
API migration

Scenario: system upgrade accompanied with a change in API.

Example from Erlang standard distribution: the regular expression library from `regexp` to `re`.

How to refactor client code to accommodate this?
Adapter **regexp to re**

```erlang
match(String, RegExp) ->
    try re:run(String, RegExp, [global]) of
       {match, Match} ->
           {Start0, Len} = lists:last(
               lists:ukeysort(
                   2, lists:append(Match)),
           Start = Start0+1,
           {match, Start, Len};
       nomatch -> nomatch
    catch
e-error:_->->
       {error, Error}=re:compile(RegExp),
       {error, Error}
    end.
```
Generate rules

From the adapter module we generate rules which “fold” the adapter into the client code, and build a refactoring to apply them automatically …
The transformation rules

A meta-rule with the template code as a case expression.

A rule with the template code as a match expression: \texttt{V@=regexp:match(S@,RE@)}

A rule with the template code an application of the old API function: \texttt{regexp:match(S@,RE@)}
Applying the rules

Step 1: apply the meta-rule.

Step 2: first apply introduce new variable refactoring for each application, then use the rule for: \( V@=\text{regexp:match}(S@, RE@) \)

Step 3: apply the third rule.

After each step cleanup to remove unused variables / expressions.
secret_path(Path, [[NewDir] | Rest], Dir) ->
case regexp:match(Path, NewDir) of
    {match, _Start, _Len} when Dir == to_be_found ->
        secret_path(Path, Rest, NewDir);
    {match, _Start, _Len} ->
        secret_path(Path, Rest, Dir);
    nomatch ->
        secret_path(Path, Rest, Dir)
end.
secret_path(Path, [[NewDir] ++ Rest], Dir) ->
case re:run(Path, NewDir, [global]) of
    {match, _Match} when Dir == to_be_found ->
        secret_path(Path, Rest, NewDir);
    {match, _Match} ->
        secret_path(Path, Rest, Dir);
    nomatch ->
        secret_path(Path, Rest, Dir)
end.
document_name(Path) ->
case regexp:match(Path, "[^/]*\$") of
    {match, Start, Len} ->
        string:substr(Path, Start, Len);
    nomatch -> "(none)"
end.
document_name(Path) ->
case re:run(Path, "[\^/]\*\$", [global]) of
   {match, Match} ->
      {Start0, Len}=lists:last(lists:ukeysort(2,Match)),
      Start = Start0 + 1,
      string:substr(Path, Start, Len);
   nomatch -> "(none)"
end.
Conclusions
Conclusions

Remove one of the barriers to adoption?

Two complementary features: API and DSL.

Go with the grain of the language.

More case studies … e.g. in RELEASE project.

Works for other languages and tools?
What next?

Refining the detailed design.

User contributions.

Application to other languages …
Questions?

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