Evolving your Projects with Wrangler

Simon Thompson and Huiqing Li
University of Kent, UK
Wrangler is a tool

Tool for refactoring Erlang programs.

Inside Emacs and ErlIDE.
Wrangler is a toolkit

Usable from the Erlang shell command line.

An open API for new refactorings …

… and a DSL for scripting complex changes.

API migration tool (e.g. `regexp` to `re`).
Wrangler is a toolset

Clone detection and module “bad smells”.

Plays with testing tools.

Laboratory: symbolic evaluation, slicing, concurrency, parallelisation.
erlang
rerlang
rangler
Wrangler
wrangler |ˈræŋgər|
noun
1 N. Amer. a person in charge of horses or other livestock on a ranch.
   • a person who trains and takes care of animals on a film set. *they had three cow wranglers to help with the scene.*
2 a person engaging in a lengthy and complicated dispute. *he was known as the wrangler for the aplomb with which he skewered the professors.*
3 (at Cambridge University) a person placed in the first class of the mathematical tripos.
Wrangler is a tool
Refactoring

“Change how the program works, but not what it does.”

Part of the programmer’s standard toolkit.

Renaming, function extraction, generalisation.
Refactoring tools

Avoid the “tedious and error-prone”.
Keep it simple.
Cover the bureaucracy.

Whole language …
… whole projects,
… plus tests, …
Demo

 Rename, function extraction, generalisation, rename again.
Wrangler is a toolkit
Wrangler is a toolkit

Wrangler is written in Erlang, all the way down, so you can extend it yourself …
Wrangler is a toolkit

... but we have given you tools to make that extension much easier to deal with.
Key idea

You know Erlang …

… you don’t want to have to learn a whole new language or compiler internals to get the job done.
An API to define a completely new refactorings from scratch … using Erlang concrete syntax, … also “code inspection”.

DSL for scripting refactorings … “on steroids” … embedded in Erlang.
What we’ve been asked for

camelCase to camel_case

Batch module renaming

Removing “bug preconditions” for Quviq.

API migration … for example, regexp to re.
Wrangler API
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}.

loop_a() ->
    receive
        stop -> ok;
        {msg, _Msg, 0} -> loop_a();
        {msg, Msg, N} ->
            body(Msg, N, "ping!~n"),
            loop_a()
    end.

body(Msg, N, Str) ->
    io:format(Str),
    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).
  end.

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

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

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

... and its context and scope.

```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

Pre-conditions for refactorings

```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}.
```

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** is used to define preconditions
- **Rules** describe transformations
- **Templates** describe expressions
- **Traversals** describe how transformations are applied
Scenario: maps in R17

Can we use maps in our project?

Look for uses of dict ... are they map-like?
Finding non **map**-like uses of **dict**

```python
find_non_map_like_dict(input_par_prompts) -> [];
find_non_map_like_dict(_Args=#{args{search_paths=SearchPaths}}) ->
  FULL_TD_TU(
    COLLECT(
      T("F@(Args@@)",
          {_File@, api_refac:start_end_loc(_This@)},
          lists:member(
            api_refac:fun_define_info(F@),
            non_map_like_dict())
      )],
    SearchPaths).
```
Results for rabbitmq-server

10 code fragments satisfying the conditions have been found.

…/rabbit_binding.erl:506.5-507.33:
…/rabbit_binding.erl:510.5-511.38:
…/rabbit_channel.erl:1213.14-1216.63:
…/rabbit_mirror_queue_slave.erl:612.10-612.78:
…/rabbit_misc.erl:759.5-759.69:
…/rabbit_msg_store.erl:1247.15-1248.56:
…/supervisor2.erl:1157.39-1157.69:
…/supervisor2.erl:1168.39-1168.69:
…/supervisor2.erl:1195.41-1195.73:
…/supervisor2.erl:1210.41-1210.73:
And **map**-like?

163 code fragments satisfying the conditions have been found.

.../delegate.erl:185.43-185.52:
.../delegate.erl:194.22-194.45:
.../delegate.erl:197.25-197.63:
.../delegate.erl:201.25-201.62:
.../delegate.erl:207.22-207.45:
.../delegate.erl:212.38-212.62:
.../delegate.erl:213.38-213.76:
.../delegate.erl:227.11-227.34:
.../delegate.erl:232.37-232.61:
......
API migration in Wrangler: maps in R17

Now we look at migrating from `dict` to `map`

We use the Wrangler API migration tool
Client #1
API #1

Client #1
API #2

Adapter
API #2
The adapter: **dict** in terms of **map**

```plaintext
is_key(Key, Dict) ->
    maps:is_key(Key, Dict).

to_list(Dict) ->
    maps:to_list(Dict).

from_list(List) ->
    maps:from_list(List).

size(Dict) ->
    maps:size(Dict).
```
The adapter: **dict** in terms of **map**

```python
is_empty(Dict) ->
    maps:size(Dict)==0.

fetch(Key, Dict) ->
    maps:get(Key, Dict).

fetch_keys(Dict) ->
    maps:keys(Dict).

erase(Key, Dict) ->
    maps:remove(Key, Dict).

store(Key, Value, Dict) ->
    maps:put(Key, Value, Dict).

update(Key, Fun, Dict1) ->
    maps:update(Key, Fun(maps:get(Key, Dict1)), Dict1).
```
Before

```erlang
predef_macros(File) ->
  Ms0 = dictionary:new(),
  true = dictionary:is_empty(Ms0),
  Ms1 = dictionary:store({atom, 'FILE'}, {none,[{string, 1, File}]}, Ms0),
  Ms2 = dictionary:store({atom, 'LINE'}, {none,[{integer, 1, 1}]}, Ms1),
  Ms3 = dictionary:store({atom, 'MODULE'}, undefined, Ms2),
  Ms4 = dictionary:store({atom, 'MODULE_STRING'}, undefined, Ms3),
  Ms5 = dictionary:store({atom, 'BASE_MODULE'}, undefined, Ms4),
  Ms6 = dictionary:store({atom, 'BASE_MODULE_STRING'}, undefined, Ms5),
  Machine = list_to_atom(erlang:system_info(mame)),
  Ms7 = dictionary:store({atom, 'MACHINE'}, {none,[{atom, 1, Machine}]}, Ms6),
  Ms8 = dictionary:store({atom, Machine}, {none,[{atom, 1, true}]}, Ms7),
  Size = dictionary:size(Ms8),
  io:format("Number of defined macros:~p\n", [Size]),
  Ms8.
```
After

```erlang
predef_macros(File) ->
    Ms0 = maps:new(),
    true = maps:size(Ms0) =:= 0,
    Ms1 = maps:put({'atom', 'FILE'}, {none, [{'string', 1, File}]}, Ms0),
    Ms2 = maps:put({'atom', 'LINE'}, {none, [{'integer', 1, 1}]}, Ms1),
    Ms3 = maps:put({'atom', 'MODULE'}, undefined, Ms2),
    Ms4 = maps:put({'atom', 'MODULE_STRING'}, undefined, Ms3),
    Ms5 = maps:put({'atom', 'BASE_MODULE'}, undefined, Ms4),
    Ms6 = maps:put({'atom', 'BASE_MODULE_STRING'}, undefined, Ms5),
    Machine = list_to_atom(erlang:system_info(machine)),
    Ms7 = maps:put({'atom', 'MACHINE'}, {none, [{'atom', 1, Machine}]}, Ms6),
    Ms8 = maps:put({'atom', Machine}, {none, [{'atom', 1, true}]}, Ms7),
    Size = maps:size(Ms8),
    io:format("Number of defined macros:~p\n", [Size]),
    Ms8.
```
Adapting to maps - how does it work?

Write adapter module. e.g. `dict_to_map.erl`

Generate API migration rules, e.g. `refac_dict_to_map.erl`

Compile the rules (to `refac_dict_to_map.beam`)

Select API migration

  → Apply API migration to current file

Input name of adaptor module `refac_dict_to_map`
Using Wrangler outside emacs/ErlIDE

Wrangler from the Erlang shell ... use module api_wrangler.erl

For example:

1> api_wrangler:start().

2> api_wrangler:rename_mod("main.erl", test, ["c:/cygwin/home/hl/test"]).

3> api_wrangler:stop().

More info: http://refactoringtools.github.io/wrangler
Wrangler is a toolset
Wrangler as a toolset

Clone detection
  Parametrisable
  Incremental
  Automated support using the DSL

Module “bad smell” detection
  Size, cycles, exports

Other inspection and refactoring functions
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}.
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.
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 \texttt{\texttt{current}(foo)} gives the ‘current’ name of an entity \texttt{foo} at any point in the refactoring.
Clone removal: top level

Transaction as a whole ... non-“atomic” components OK.

Not just an API: \texttt{atomic} etc. modify interpretation of what they enclose ...

\texttt{atomic}\texttt{[?interactive( RENAME FUNCTION )
  \texttt{refac}( RENAME ALL VARIABLES OF THE FORM NewVar* )
  \texttt{repeat_interactive}( SWAP ARGUMENTS )
  \texttt{if_then}( EXPORT IF NOT ALREADY )
  \texttt{non_atomic}( FOLD INSTANCES OF THE CLONE )\texttt{]}).}
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])
Refactoring tools for functional languages

SIMON THOMPSON and HUIQING LI
School of Computing, University of Kent, Kent, UK
(e-mail: {s.j.thompson, h.li}@kent.ac.uk)

Abstract

Refactoring is the process of changing the design of a program without changing what it does. Typical refactorings, such as function extraction and generalisation, are intended to make a program more amenable to extension, more comprehensible and so on. Refactorings differ from other sorts of program transformation in being applied to source code, rather than to a ‘core’ language within a compiler, and also in having an effect across a code base, rather than to a single function definition, say. Because of this, there is a need to give automated support to the process. This paper reflects on our experience of building tools to refactor functional programs written in Haskell (HaRe) and Erlang (Wrangler). We begin by discussing what refactoring means for functional programming languages, first in theory, and then in the context of a larger example. Next, we address system design and details of system implementation as well as contrasting the style of refactoring and tooling for Haskell and Erlang. Building both tools led to reflections about what particular refactorings mean, as well as requiring analyses of various kinds, and we discuss both of these. We also discuss various extensions to the core tools, including integrating the tools with test frameworks; facilities for detecting and eliminating code clones; and facilities to make the systems extensible by users. We then reflect on our work by drawing some general conclusions, some of which apply particularly to functional languages, while many others are of general value.

http://journals.cambridge.org/repo_A90unxgK
Wrangelska palatset
Wrangelska Palace


Fältären Carl Gustav Wrangel satte i gång palatsbygget 1652. Fyra år tidigare hade han fått tomten i belöning för sina insatser i 30-åriga kriget av drottning Kristina. Här stod då ett stenhus i två våningar som Lars Sparre låtit bygga 20 år tidigare. I ena hörnet ner mot stranden ingick ett kanontorn från Gustav Vasas dagar.


For the coronation of Karl XII in 1697 the palace was turned into a hall of splendour, where the king sat on a silver throne. The Wrangel Palace had become the temporary residence of the Royal Family. Since 1755 it is used by Svea Hovrätt Court of Appeal.

Count Carl Gustav Wrangel started constructing the palace in 1652. Four years earlier Queen Kristina had granted him the land for his exploits in the Thirty Years’ War. A 20-year-old, two-storey stone house stood here, featuring at one corner facing the shore a cannon turret from 1530.

Wrangel enlisted architects Jean De la Vallée and Nicodemus Tessin the Elder to turn the stone house into a magnificent palace. The façades were decorated in sandstone and the most imposing side overlooking the water was given balance through a second round tower. Facing the square, two wings were added, ending in square towers. Wrangel died in 1676, the palace still incomplete, but it was the biggest and grandest dwelling in the town.

In 1697 the castle Tre Kronor burned down, and the palace became home to the Royal Family until 1754. In 1755 several public bodies moved in, including Svea Hovrätt, founded in 1614. Much was destroyed by fire in 1802. In 1804, the palace was redesigned to drawings by Carl Christoffer Gjörwell, giving it today’s appearance. Alterations were carried out in 2004 to stringent specifications on security and public accessibility.
Getting involved

https://github.com/RefactoringTools/Wrangler
Questions?