Refactoring functional programs: past and future

Simon Thompson and Huiqing Li
University of Kent

COW 32

Outline

Refactoring and functional programming.

Some tools that we have built.

Examples, examples, examples.

Some ideas for an agenda for the future.

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.
```

Let’s turn this into a function

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.
```

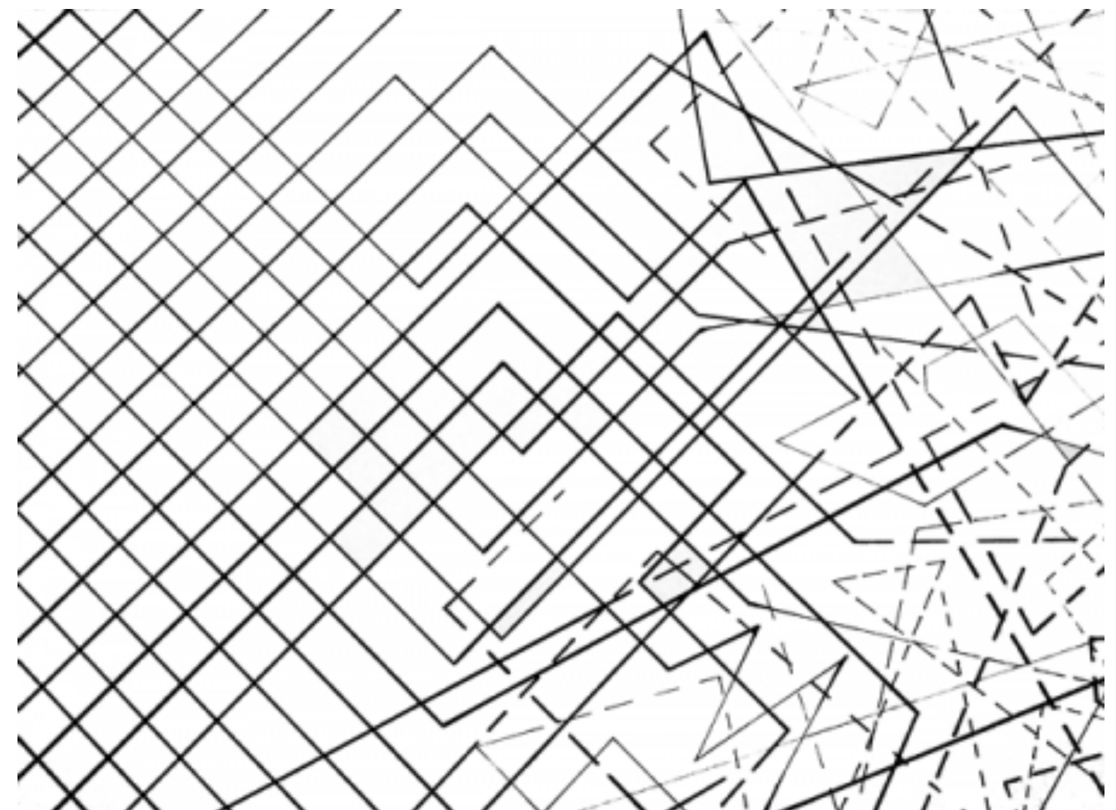
```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
     end.

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.



Refactoring *functional* programs

Highly expressive expression language … Tidier, HLint, … .

More abstractions available:

can wrap side-effecting code in a closure;

can abstract over functionality, and not just data.

Semantics "cleaner" even if not fully formal.

Potentially more trustworthy:

semantics and implementation language.

How to refactor?

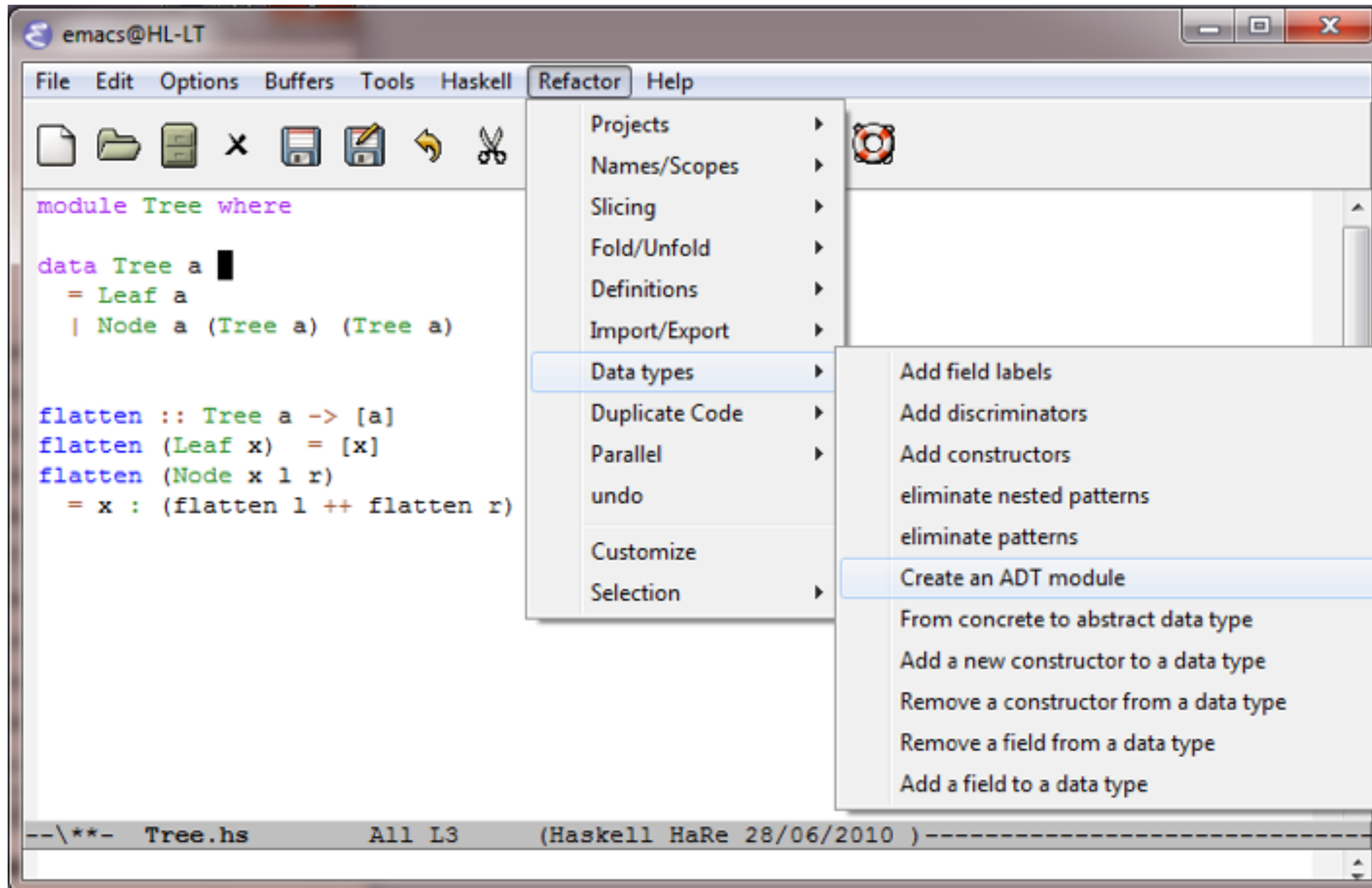
By hand … using an editor.

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

Tool-supported.

- Handle atoms, types, names, side-effects, …
- Scalable to large-code bases: module-aware.
- Integrated with tests, macros, ...

HaRe



Aquamacs File Edit Options Tools Wrangler Erlang Window Help

New Open Recent Save Print

scratch | test_camel_case.erl

```erlang
-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.

Wrangler menu

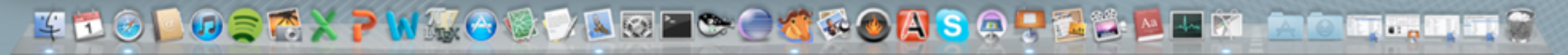
- Refactor ▶
- Inspector ▶
- Undo ^C ^W _
- Similar Code Detection ▶
- Module Structure ▶
- API Migration ▶
- Skeletons ▶
- Customize Wrangler
- Version

Refactor submenu

Item	Shortcut
Rename Variable Name	^C ^W R V
Rename Function Name	^C ^W R F
Rename Module Name	^C ^W R M
Generalise Function Definition	^C ^G
Move Function to Another Module	^C ^W M
Function Extraction	^C ^W N F
Introduce New Variable	^C ^W N V
Inline Variable	^C ^W I
Fold Expression Against Function	^C ^W F F
Tuple Function Arguments	^C ^W T
Unfold Function Application	^C ^W U
Introduce a Macro	^C ^W N M
Fold Against Macro Definition	^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/Remove Menu Items ▶	

gen_refac Refacs submenu

- Swap Function Arguments
- Specialise A Function
- Remove An Import Attribute
- Remove An Argument
- Keysearch To Keyfind
- Apply To Remote Call
- Add To Export
- Add An Import Attribute



HaRe and 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 the language and apply the tool to itself.

Wrangler

Clone detection and removal

Module structure improvement

DSL for composite refactorings

API: define new refactorings

Basic refactorings: structural, macro, process and test-framework related

Examples

Examples

Basic refactorings …

… and some of their complexity.

Helping the user …

… clone detection, module structure.

Working in specialised domains …

… web services, testing frameworks.

Extensibility …

… an API and a DSL.

Getting started … what did we mean?

Generalisation … in Haskell

```
f x y z = length ((2:x) ++ []) +
          length ((True:y) ++ []) +
          length ((3:z) ++ [])
```

Generalise over the `[]`.

Generalisation … in Haskell

```
f x y z = length ((2:x) ++ []) +
          length ((True:y) ++ []) +
          length ((3:z) ++ [])
```

Generalise over the `[]`.

What do you mean: one, all, some?

```
f x y z w = length ((2:x) ++ w) +
            length ((True:y) ++ w) +
            length ((3:z) ++ [])
```

What is the type of `w`?

Generalisation … in Erlang

```
f([]) -> ok;
```

Call `f([2,2,3])`

```
f([X|Xs]) ->
      io:format(“~p~t”,[X]),
      f(Xs).
```

Generalise over `io:format(“~p~t”,[X])`.

Generalisation … in Erlang

```
f([]) -> ok;
```

Call `f([2,2,3])`

```
f([X|Xs]) ->
      io:format("~p~t",[X]),
      f(Xs).
```

Generalise over `io:format("~p~t",[X])`.

What about the side-effect and the free variable?

```
f([],h) -> ok;
```

Call `f([2,2,3], fun(X) -> io:format("~p~t",[X]) end)`

```
f([X|Xs],h) ->
      h(X),
      f(Xs).
```

Lifting definitions … in Haskell

```
h x = x + g x
      where
       g x = x + con
       con = 37
```

Lift `g` to be a top-level definition. What about `con`?

Lifting definitions … in Haskell

```
h x = x + g x
      where
       g x = x + con
       con = 37
```

Lift g to be a top-level definition. What about con?

Lambda lift?

```
h x = x + g con x
      where
       con = 37

g con x = x + con
```

Lifting definitions … in Haskell

```
h x = x + g x
      where
       g x = x + con
       con = 37
```

Lift g to be a top-level definition. What about con?

Localise before lifting?

```
h x = x + g x

g x = x + con
       where
       con = 37
```

Lifting definitions … in Haskell

```
h x = x + g x
      where
       g x = x + con
       con = 37
```

Lift `g` to be a top-level definition. What about `con`?

Lift all dependents?

```
h x = x + g x

g x = x + con

con = 37
```

Being informed … in particular domains

Clone removal

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)---

- Rename function
- Rename variables
- Reorder variables
- Add to export list
- Fold* against the def.

Extending it yourself

Extensibility: API + DSL

API

Describe entirely new ‘atomic’ refactorings from scratch.

e.g. swap args, delete argument.

We assume you know Erlang, but not internals of the syntax.

DSL

A language to script composite refactorings on top of simpler ones.

e.g. remove clone, migrate from old to new API.

We *embed* in Erlang, to use the language in the “scripts”.

API: templates and rules … in Erlang

```
?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 …
```

Wrangler API

Context
available for
pre-conditions

Traversals
describe how
rules are applied

Rules describe transformations

Templates describe expressions

Aquamacs File Edit Options Tools Wrangler Erlang Window Help

(0:15) 09:32

New Open Recent Save Print

scratch test_camel_case.erl

```
-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)

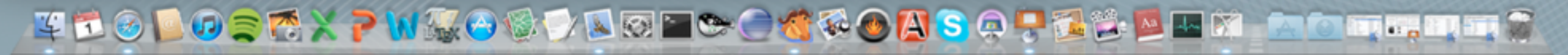
Wrangler started.

- Refactor ▶
- Inspector ▶
- Undo ^C ^W _
- Similar Code Detection ▶
- Module Structure ▶
- API Migration ▶
- Skeletons ▶
- Customize Wrangler
- Version

Rename Variable Name	^C ^W R V
Rename Function Name	^C ^W R F
Rename Module Name	^C ^W R M
Generalise Function Definition	^C ^G
Move Function to Another Module	^C ^W M
Function Extraction	^C ^W N F
Introduce New Variable	^C ^W N V
Inline Variable	^C ^W I
Fold Expression Against Function	^C ^W F F
Tuple Function Arguments	^C ^W T
Unfold Function Application	^C ^W U
Introduce a Macro	^C ^W N M
Fold Against Macro Definition	^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/Remove Menu Items ▶	

- Swap Function Arguments
- Specialise A Function
- Remove An Import Attribute
- Remove An Argument
- Keysearch To Keyfind
- Apply To Remote Call
- Add To Export
- Add An Import Attribute

Digital Theses.doc Macintosh HD TTFP errata.html simonthompson papers info O'Reilly OTP book shared Review 2011 REF



DSL … not just a script

Tracking changing names and positions.

Generating refactoring commands.

Dealing with failure.

User control of execution.

Deals with the pragmatics of composition, rather than the theory.

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).
```

Aquamacs File Edit Options Tools Wrangler Erlang Window Help

New Open Recent Save Print

scratch test_camel_case.erl

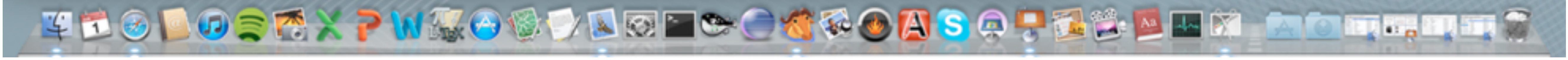
```
-module(test_camel_case).

-export([thisIsAFunction/2,
         this_is_a_function/2,
         thisIsAnotherFunction/

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

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

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

- Refactor ▶
- Inspector ▶
- Undo ^C ^W _
- Similar Code Detection ▶
- Module Structure ▶
- API Migration ▶
- Skeletons ▶
- Customize Wrangler
- Version

Rename Variable Name	^C ^W R V
Rename Function Name	^C ^W R F
Rename Module Name	^C ^W R M
Generalise Function Definition	^C ^G
Move Function to Another Module	^C ^W M
Function Extraction	^C ^W N F
Introduce New Variable	^C ^W N V
Inline Variable	^C ^W I
Fold Expression Against Function	^C ^W F F
Tuple Function Arguments	^C ^W T
Unfold Function Application	^C ^W U
Introduce a Macro	^C ^W N M
Fold Against Macro Definition	^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 ▶	refac_camel_case
Apply Adhoc Refactoring	
Apply Composite Refactoring	
Add/Remove Menu Items ▶	

-:--- test_camel_case.erl All (13,0) (Erlang EXT Flymake)

Wrangler started.



Clone removal in the DSL

Transaction as a whole … non-transactional components OK.

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

```
?transaction(
     [?interactive( RENAME FUNCTION )
      ?refac_( RENAME ALL VARIABLES OF THE FORM NewVar*)
      ?repeat_interactive( SWAP ARGUMENTS )
      ?if_then( EXPORT IF NOT ALREADY )
      ?non_transaction( FOLD INSTANCES OF THE CLONE )
     ]).
```

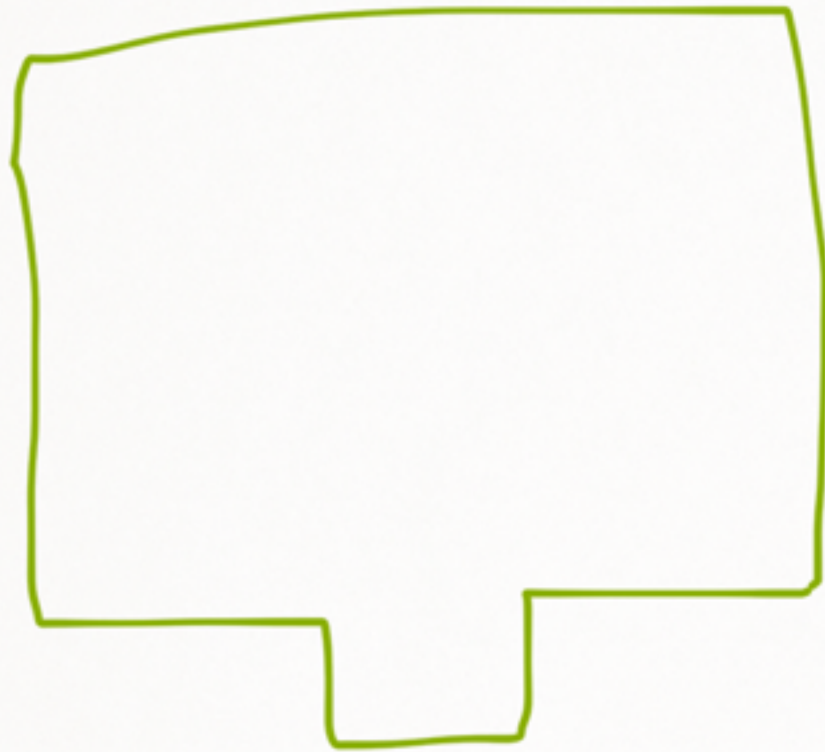
API migration

Scenario: system upgrade accompanied with a change in API.

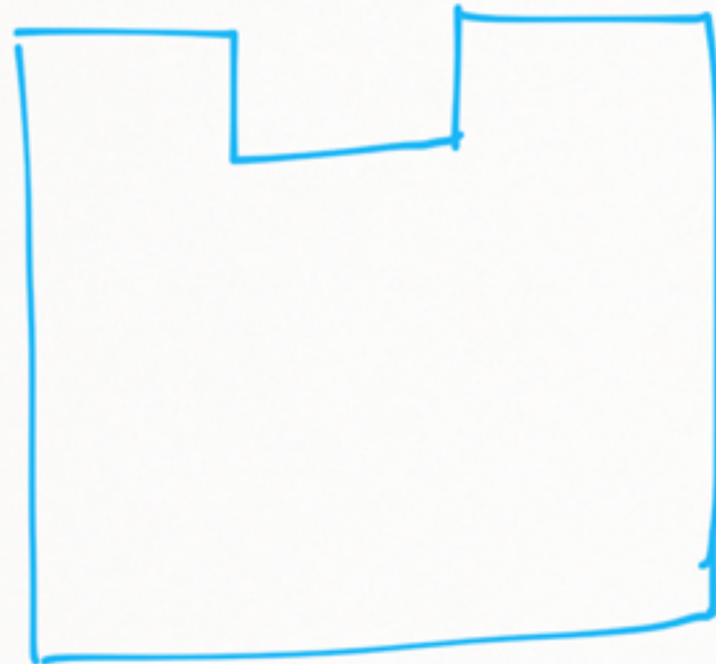
Example from Erlang standard distribution: the regular expression library from `regexp` to `re`.

How to refactor client code to accommodate this?

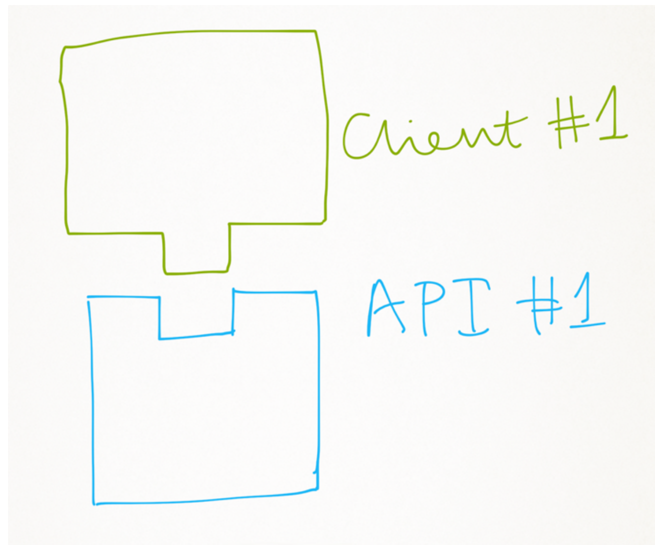
Case study in the use of the API + DSL.



Client #1

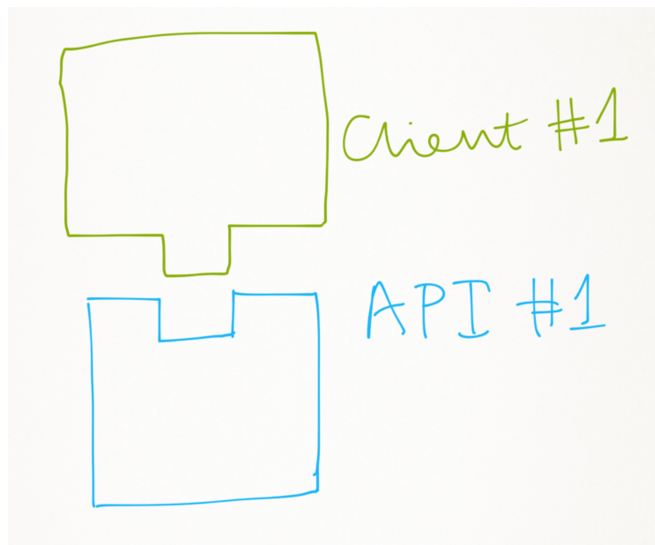


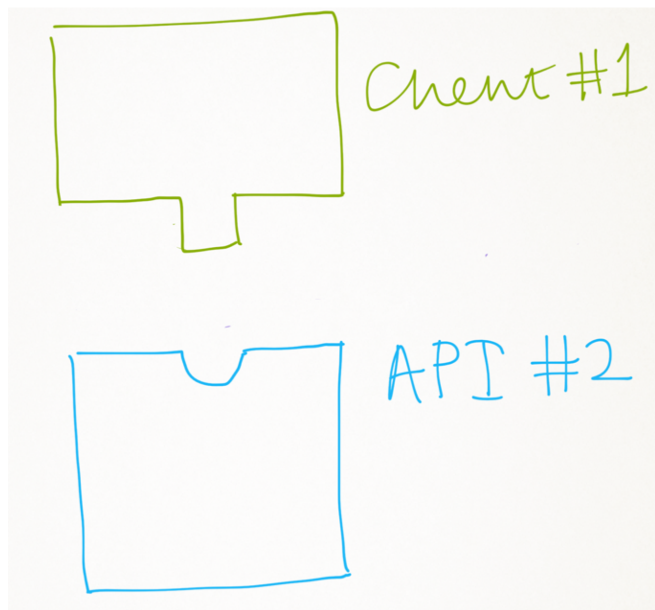
API #1



Client #1

API #2

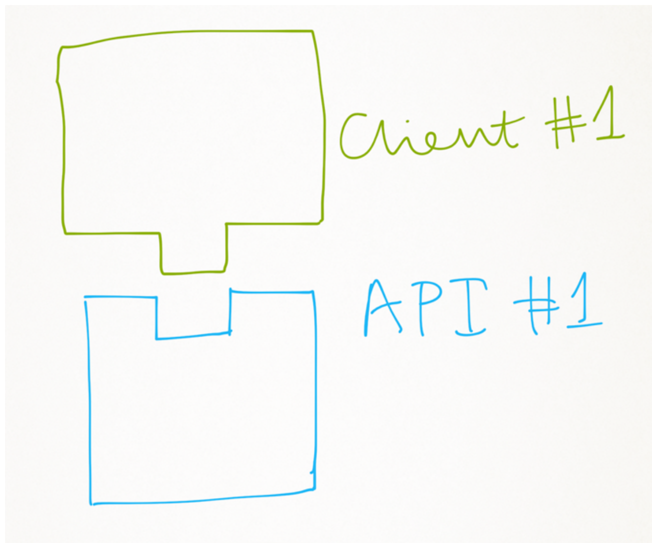


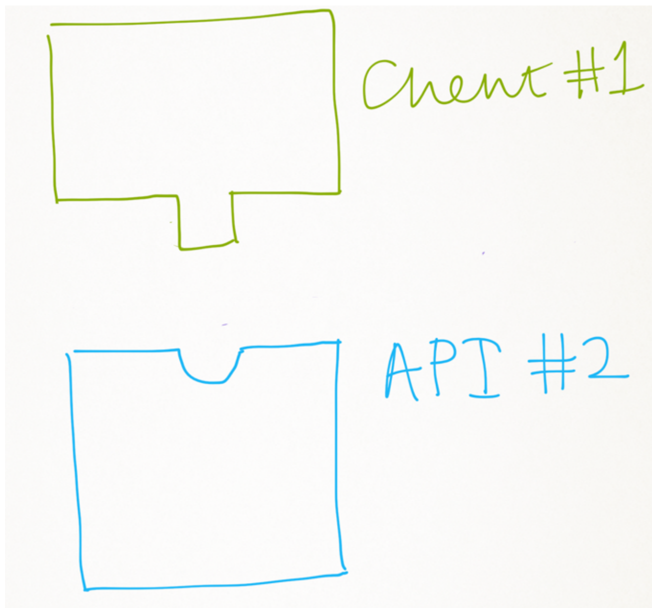


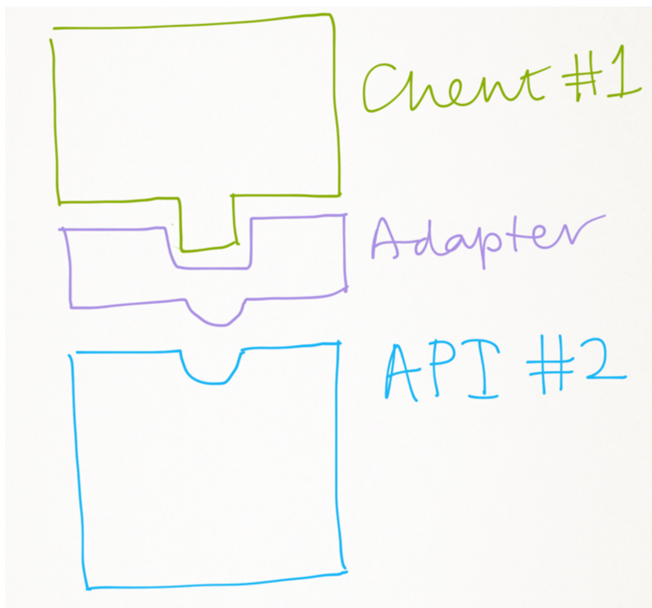
Client #1

Adapter

API #2







Client #2

API #2

Looking forward

“Why should I trust my code to your tool?”

Benefit ≫ risk: removing bug preconditions

Scenario: building Erlang models for C code at Quviq AB.

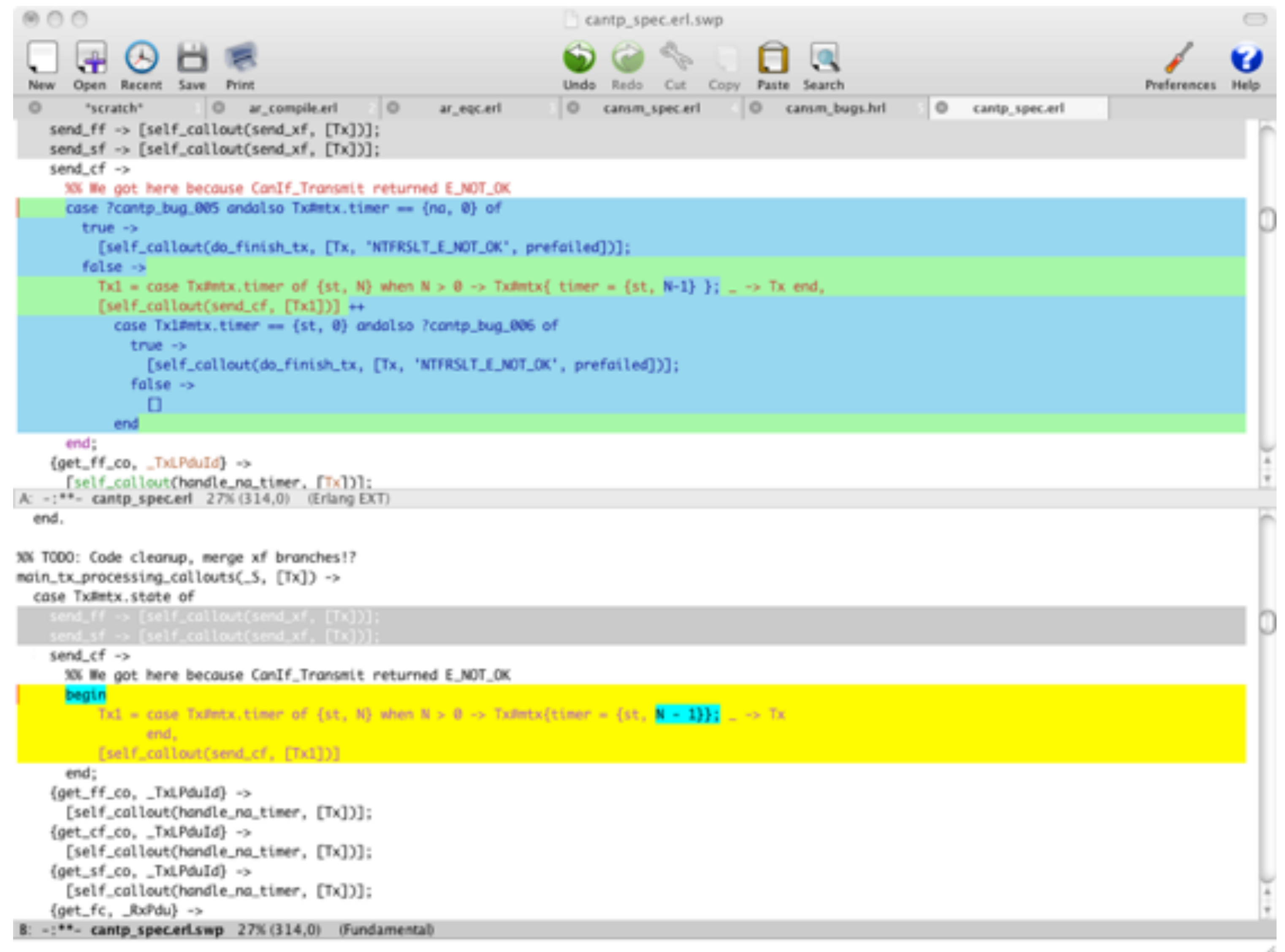
For buggy code, want to avoid hitting the same bugs all the time.

Add bug precondition macros …

… but want to remove in delivered code.

DSL + API.

And you can see the changes …



The appearance has changed!

```
my_list() ->
    [ foo,
      bar,
      baz,
      wombat
    ]

my_funny_list() ->
    [ foo
      ,bar
      ,baz
      ,wombat
    ]
```

```
{v1, v2, v3}

{v1,v2,v3}

f (g x y)

f $ g x y
```

```
data MyType = Foo |
              Bar |
              Baz

data HerType = Foo
             | Bar
             | Baz
```

Preserving meaning

What are we preserving?

Where are we preserving it?

Individual results or the refactoring tool itself?

Equivalences

Testing equivalence: $\forall$ test data [finite]

PBT equivalence: $\forall$ random test data [finite, but unbounded]

Extensional equivalence: $\forall$ input data [infinite]

(Annotated) abstract syntax tree (with some quotient?)

Textual

Question: varieties of $\downarrow$: may be happy to converge on *more* inputs?

tool or results

test or verify

×	

tool or results

test or verify

tool or results

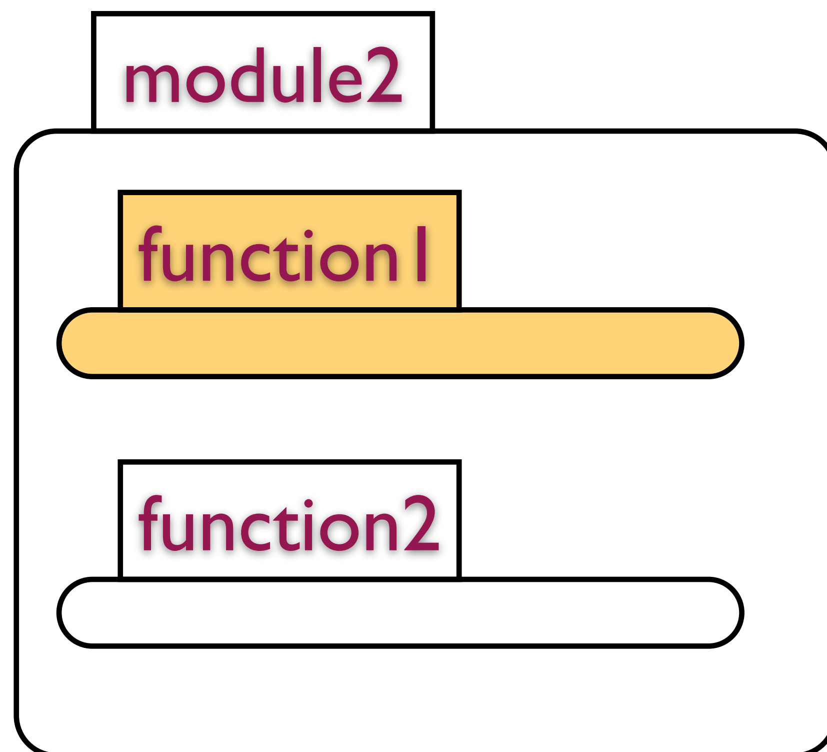
test or verify

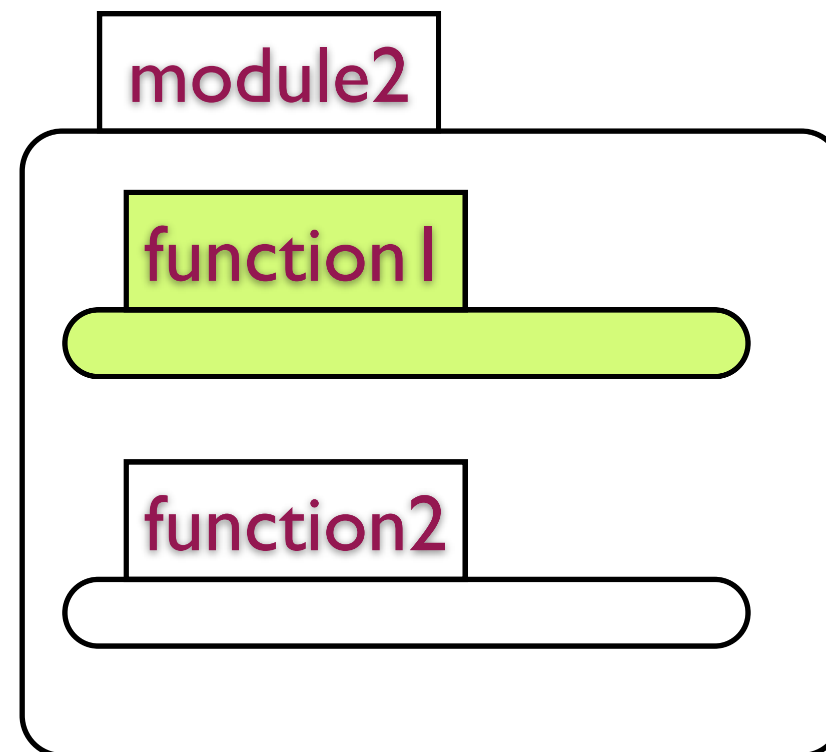
Testing two refactoring tools

Compare the results of tool1 and tool2 …

… either by testing both, or directly comparing the code / ASTs.

Similar to compiler comparisons and Eclipse *vs* NetBeans (Dig *et al*).



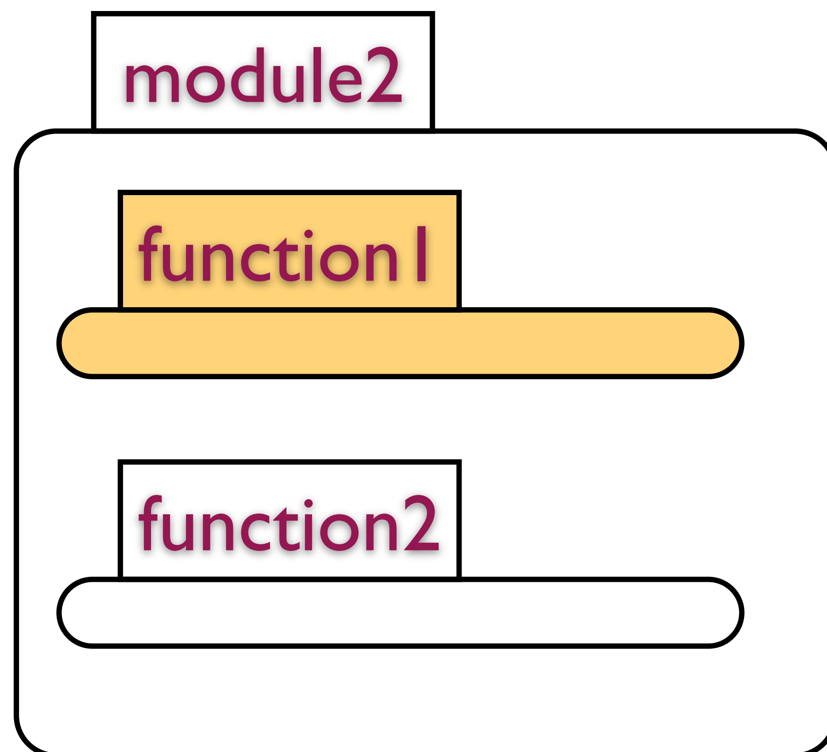


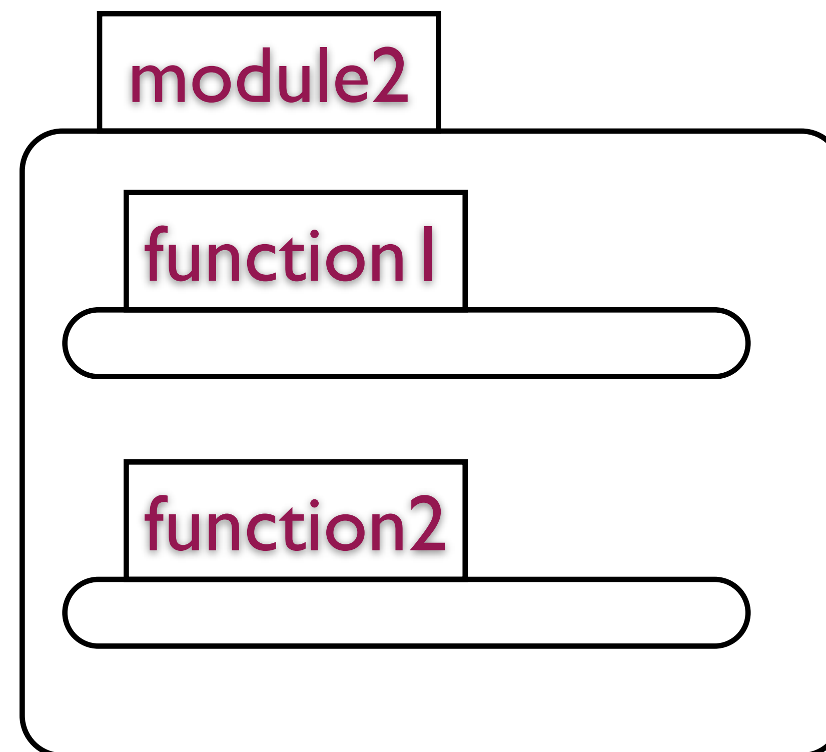
Testing one tool

Compare the results of function1 and function1 (unmodified) …

… using existing unit tests, or randomly-generated inputs

… could compare ASTs as well as behaviour (in former case).



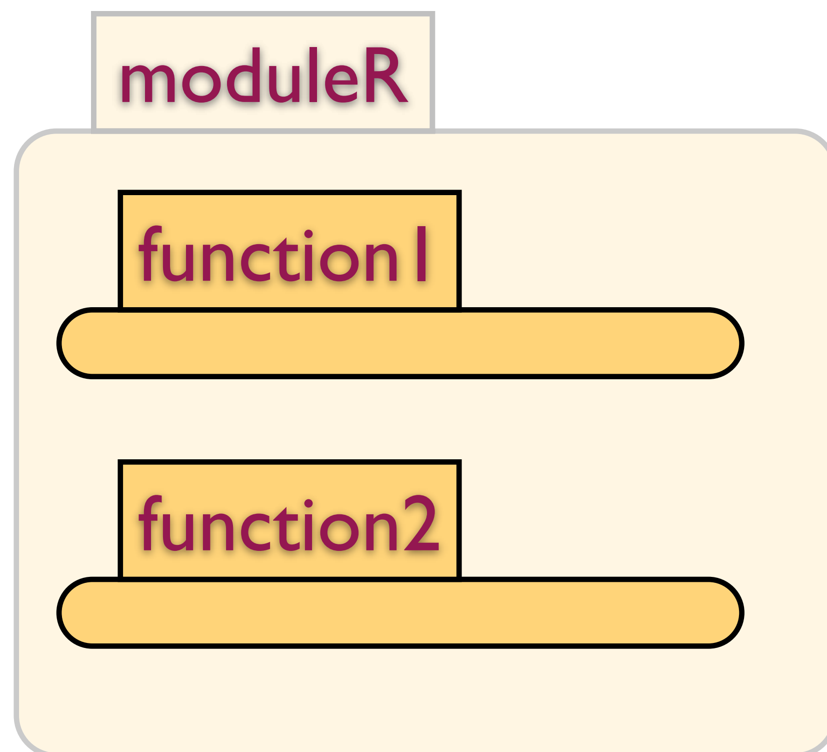


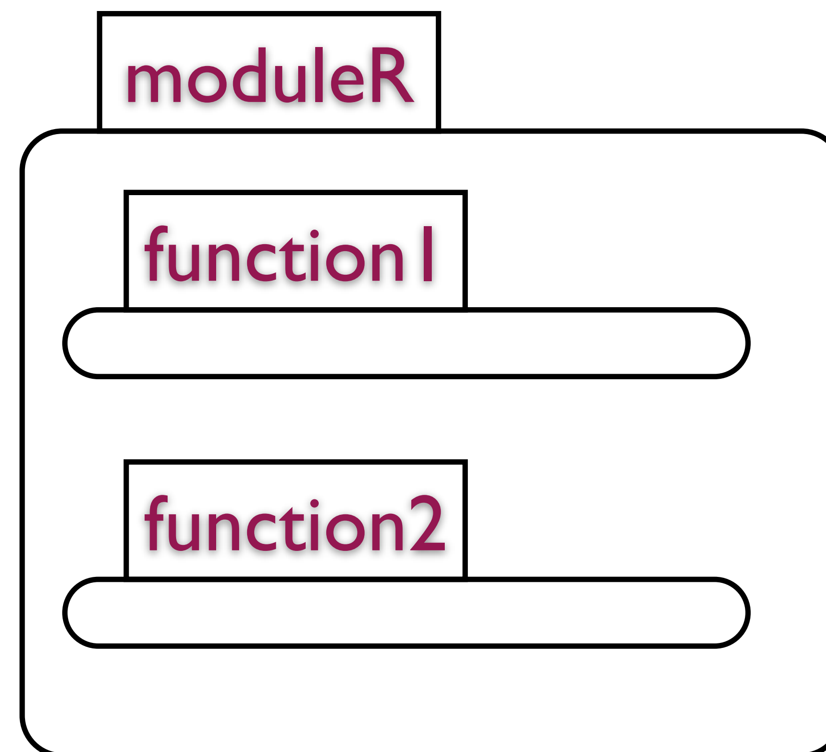
Fully random

Generate random modules,

… generate random refactoring commands,

… and check ≡ with random inputs. (w/ Drienyovszky, Horpácsi).





tool or results

test or verify

Tool verification (with Nik Sultana)

$$\forall p.\ (Q\,p) \longrightarrow (T\,p) \simeq p$$

Deep embeddings of small languages:

… potentially name-capturing λ-calculus

… PCF with unit and sum types.

Isabelle/HOL: LCF-style secure proof checking.

Formalisation of meta-theory: variable binding, free / bound variables, capture, fresh variables, typing rules, etc …

… principally to support pre-conditions.

Variable capturing substitution

$$
\begin{aligned}
\varepsilon[M/x] &\stackrel{\text{def}}{=} \varepsilon \\
(y := N)[M/x] &\stackrel{\text{def}}{=} \text{if } x = y \text{ then } y := N \\
&\quad\ \text{else } y := (N[M/x]) \\
(D_1 \parallel D_2)[M/x] &\stackrel{\text{def}}{=} \text{if } x \in \mathit{DVTopd}\,(D_1 \parallel D_2) \\
&\quad\ \text{then } (D_1 \parallel D_2) \\
&\quad\ \text{else } (D_1[M/x] \parallel D_2[M/x]) \\
i[M/x] &\stackrel{\text{def}}{=} \text{if } x = i \text{ then } M \text{ else } i \\
(\lambda i.N)[M/x] &\stackrel{\text{def}}{=} \text{if } x = i \text{ then } \lambda i.N \\
&\quad\ \text{else } \lambda i.(N[M/x]) \\
(N \cdot N')[M/x] &\stackrel{\text{def}}{=} (N[M/x]) \cdot (N'[M/x]) \\
(\mathit{letrec}\, D \ \mathit{in}\ N)[M/x] &\stackrel{\text{def}}{=} \text{if } x \in \mathit{DVTopd}\,(\mathit{letrec}\, D \ \mathit{in}\ N) \\
&\quad\ \text{then } (\mathit{letrec}\, D \ \mathit{in}\ N) \\
&\quad\ \text{else } \mathit{letrec}\,(D[M/x]) \ \mathit{in}\ (N[M/x])
\end{aligned}
$$

tool or results

test or verify

Automatically verify instances of refactorings

Prove the equivalence of the particular pair of functions / systems using an SMT solver …

… SMT solvers linked to Haskell by `Data.SBV` (Levent Erkok).

Manifestly clear what is being checked.

The approach delegates trust to the SMT solver …

… can choose other solvers, and examine counter-examples.

Also possible for Erlang using e.g. McErlang model checker.

Example: renaming

```
module Before where

h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

Example: renaming

```haskell
module Before where

h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

```haskell
module After where

h :: Integer->Integer->Integer

h x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
-- f can be treated as an uninterpreted symbol

f = uninterpret "f"

-- Properties

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
-- f can be treated as an uninterpreted symbol

f = uninterpret "f"

-- Properties

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```
*Refac2> propertyk
Q.E.D.
*Refac2> propertyh
Q.E.D.
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)
         where
         g z = z*z

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)
         where
         g z = z*z

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
f = uninterpret "f"

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)
         where
         g z = z*z

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
f = uninterpret "f"

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```
*Refac2> propertyk
Q.E.D.
*Refac2> propertyh
Falsifiable. Counter-example:
  s0 = 0 :: SInteger
  s1 = -1 :: SInteger
```

“How do I refactor my data representation?”

Changing data representations

Modify the implementation of a particular type (synonym).

But don't modify all occurrences of `(Int,Bool)`, … scope issue.

```
type Rep = (Int,Bool)

f :: Rep -> Int
f (n,_) = n + 42

g :: (Int,Bool) -> Rep
g (n,b) =
  if b then (n,b) else (-n,b)

h :: Rep -> Bool
h = snd
```

```
type Rep = (Bool,Int)

f :: Rep -> Int
f (_,n) = n + 42

g :: (Int,Bool) -> Rep
g (n,b) =
  if b then (b,n) else (b,-n)

h :: Rep -> Bool
h = snd . flip
```

Changing data representations

Where does it get interesting?

Introducing monad or applicative, e.g.

`Int -> (x,Int)` to `State Int x`

Introducing monad transformers.

Non-isomorphic representations.

Reactive extensions.

Going to OTP, distributed, replicated, supervised …

“Can I apply this to GHC Haskell?”

"How do I refactor my Erlang + JavaScript?"

“Who cares about text files these days?”

“How do I parallelise this code?”

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