Building trustworthy refactoring tools

Simon Thompson, University of Kent, UK

Joint work with Thomas Arts, Dániel Drienyovszky, Dániel Horpácsi, Huiqing Li and Nik Sultana
Why should I trust my code to your refactoring tool?
Outline

What we do … and how we do it
Psychological, pragmatic and technical
A range of equivalences
Testing … property-based testing
Verification … manual and automated
System-level and program-level
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

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

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

- Clone detection and removal
- Module structure improvement
- DSL for composite refactorings
- API: define new refactorings
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 ...
Clone removal

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

```plaintext
?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 )
  ]).
```
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) ->
            X + Y.
-module(test_camel_case).
-export([thisIsAFunction/2, this_is_a_function/2, thisIsAnotherFunction/1]).

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

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

thisIsAnotherFunction(X, Y) ->
  X+Y.
Under the hood

- Parse
- Analyse
- Transform
- Output

- text
- AST
- AAST
- AAST
- text
Why should I trust my code to your refactoring tool?
Psychological and social issues

Open Source … confidence in the code … other committers.

The benefits outweigh the risk / cost …

… might even be OK to introduce some faults?

Openness of the system …

… you can check the changes that a refactoring makes,

… and for the DSL can see which refactorings performed.
Benefit vs. 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 …
Pragmatic issues

GHC vs Haskell standards vs other Haskell implementations

Editor and IDE integration

Wider integration: comments, makefiles, tests, …

*It does exactly what I said (or want?)* … API and DSL.
Technical

Meaning has been preserved.

Appearance has been preserved.
The appearance hasn’t changed

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

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

{v1, v2, v3}

{v1, v2, v3}

data MyType = Foo | Bar | Baz

data HerType = Foo | Bar | Baz

f (g x y)
f $ g x y
Preserving appearance

Preserve precisely parts not touched.

Pretty print … or use lexical details.

Learn layout for synthesised code from existing codebase.
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?
<table>
<thead>
<tr>
<th>tool or results</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>test or verify</td>
</tr>
</tbody>
</table>
Testing
Testing the results of applying the tool …

Regression tests (and properties) for the system …

… and at module and function level (modulo refactoring).
test or verify

tool or results
System, module and function

main

module1

module2

function1

function2

should be preserved

may be preserved

probably not preserved
System + unit testing … refactor tests too

main

module 1

module 2

function 1

function 2

should be preserved

should be preserved

should be preserved
System + PBT ... refactor properties too

- main
  - module1
    - function1
    - function2
  - module2
    - should be preserved
    - should be preserved
    - should be preserved
… or testing the refactoring tool itself.

Generate programs as test data for the tool …

… together with refactorings and test data for the programs.
<table>
<thead>
<tr>
<th>tool or results</th>
<th>test or verify</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
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 $\equiv$ with random inputs. (w/ Drienyovszky, Horpácsi).
Verification
Verification

Tool-level verification for little languages …

… or for full scale tools (re-)using implemented meta-theory?

Individual verifications: proof or counterexample.
test or verify

<table>
<thead>
<tr>
<th>tool or results</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tool verification (with Nik Sultana)

\[ \forall p. (Q 
\phantom{p}) \longrightarrow (T 
\phantom{p}) \simeq p \]

Deep embeddings of small languages:

... potentially name-capturing \( \lambda \)-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

\[ \varepsilon[M/x] \quad \text{def} \quad \varepsilon \]

\[ (y := N)[M/x] \quad \text{def} \quad \text{if } x = y \text{ then } y := N \text{ else } y := (N[M/x]) \]

\[ (D_1 \parallel D_2)[M/x] \quad \text{def} \quad \text{if } x \in DVTopd (D_1 \parallel D_2) \text{ then } (D_1 \parallel D_2) \text{ else } (D_1[M/x] \parallel D_2[M/x]) \]

\[ i[M/x] \quad \text{def} \quad \text{if } x = i \text{ then } M \text{ else } i \]

\[ (\lambda i.N)[M/x] \quad \text{def} \quad \text{if } x = i \text{ then } \lambda i.N \text{ else } \lambda i.(N[M/x]) \]

\[ (N \cdot N')[M/x] \quad \text{def} \quad (N[M/x]) \cdot (N'[M/x]) \]

\[ (\text{letrec } D \text{ in } N)[M/x] \quad \text{def} \quad \text{if } x \in DVTopd (\text{letrec } D \text{ in } N) \text{ then } (\text{letrec } D \text{ in } N) \text{ else } \text{letrec } (D[M/x]) \text{ in } (N[M/x]) \]
Inductive definition of evaluation

\[ Fresh(z, M) \quad \frac{\lambda x. M \simeq \lambda z. M[z/x]}{(\alpha)} \]

\[ \neg \text{Captures}(N, M) \quad \frac{(\lambda x. M) N \simeq M[N/x]}{(\beta)} \]

\[ x \notin \text{FV}(M) \quad \frac{\lambda x. (M \cdot x) \simeq M}{(\eta)} \]

\[ M \simeq M \quad \text{(REFL)} \quad N \simeq M \quad \text{(SYMM)} \]

\[ M \simeq M' \quad M' \simeq N \quad \frac{M \simeq N}{(\text{TRAN})} \]
Extract a (local) definition

The composition of four steps

1. `letrec f := M in L` is the original expression, and is changed to
2. `letrec f := letrec g := N in M[g:N] in L` by “declare a definition”, then to
3. `letrec g := N in letrec f := letrec g := N in M[g:N] in L` using “add a redundant definition”, and finally to
4. `letrec g := N in letrec f := M[g:N] in L` by using “demote a definition”.

Extract a (local) definition … formally

\[ g \notin FV L \land \\
\neg Rec (g := N) \land \\
g \# (f := M) \land \\
N \subseteq L M \land \\
\neg Captures fix f \land \\
\neg Captures L g \land \\
\neg Captures N f \land \\
\neg Captures L M \land \\
\neg Captures N M \land \\
\neg Captures (letrec f := (letrec g := N in M) in L) N \land \\
\neg Captures L (M[g:N]) \land \neg Captures N (M[g:N]) \rightarrow \\
\]

\[\text{letrec } f := M \text{ in } L \approx \text{letrec } g := N \text{ in } (\text{letrec } f := M[g:N] \text{ in } L)\]
PCF + union: expand type example

If

- $\Gamma \vdash N :: S \land \Gamma \vdash x :: T \land \Gamma, y : T' \vdash L :: T \land \Gamma \vdash M :: T$
- $\neg \text{Captures } N \langle x' \leftarrow x' \rangle x \langle y \Rightarrow L \rangle \land \neg \text{Captures } N \ M \land \neg \text{Captures } L \ M$
- $x' \notin FV \ M \land y \notin FV \ M \land x \notin FV \ L$

Then

$\Gamma \vdash \text{let } x : T := M \ \text{in } N \ \simeq$

$let \ x : T + T' := \text{inL}_{T+T'} \ M \ \text{in } N[\langle x' \leftarrow x' \rangle x \langle y \Rightarrow L \rangle / x] :: S$
Full tool verification revisited

Tool-level verification for full scale tools?

This requires, at least:

- Meta-theory for a real language
- Semantics for a real language
Idea (with Nik Sultana)

Prove the equivalence of a class of pairs of functions in a theorem prover …

… and extract the transformation function as the refactoring using Haskell extraction facilities.

Again, will be a proof for a small language …

… but what about a refactoring (for dependent types?) written in a dependently typed language like Agda?
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

module Before where

h :: Integer->Integer->Integer
h x y = g y + f (g y)

f :: Integer->Integer
f x = x + 1

g :: Integer->Integer
gh x = 3*x + f x
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

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
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV

h :: Integer->Integer->Integer
h x y = g y + f (g y)

g :: Integer->Integer

f x = 3*x + f x
{-# LANGUAGE ScopedTypeVariables #-}
module RefacProof where
import Data.SBV

h :: Integer->Integer->Integer
h x y = g y + f (g y)

g :: Integer->Integer
g x = 3*x + f x

h' :: Integer->Integer->Integer
h' x y = k y + f (k y)

k :: Integer->Integer
k x = 3*x + f x
{-# LANGUAGE ScopedTypeVariables #-}
module RefacProof where
import Data.SBV

h :: Integer->Integer->Integer
h x y = g y + f (g y)

k :: Integer->Integer
k x = 3*x + f x

h' :: Integer->Integer->Integer
h' x y = k y + f (k y)

f = uninterpret "f"

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
\( h :: \text{Integer} \to \text{Integer} \to \text{Integer} \)
\[
h \ x \ y = g \ y + f (g \ y)
\]
\( g :: \text{Integer} \to \text{Integer} \)
\[
g \ x = 3\times x + f \ x
\]

\( h' :: \text{Integer} \to \text{Integer} \to \text{Integer} \)
\[
h' \ x \ y = k \ y + f (k \ y)
\]
\( k :: \text{Integer} \to \text{Integer} \)
\[
k \ x = 3\times x + f \ x
\]

-- \( f \) can be treated as an uninterpreted symbol

\( f = \text{uninterpret} \ "f" \)

-- Properties

\[
\text{property}\_k = \text{prove} \ \ (x::\text{SInteger}) \to g \ x .== k \ x
\]
\[
\text{property}\_h = \text{prove} \ \ (x::\text{SInteger}) \ (y::\text{SInteger}) \to h \ x \ y .== h' \ x \ y
\]

*Refac2* \(\geqslant\) property\_k

Q.E.D.

*Refac2* \(\geqslant\) property\_h

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
\[ h \; :: \; \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]

\[ h \; x \; y = g \; y + f \; (g \; y) \]

where

\[ g \; z = z \times z \]

\[ g \; :: \; \text{Integer} \rightarrow \text{Integer} \]

\[ g \; x = 3 \times x + f \; x \]

\[ h' \; :: \; \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]

\[ h' \; x \; y = k \; y + f \; (k \; y) \]

where

\[ g \; z = z \times z \]

\[ k \; :: \; \text{Integer} \rightarrow \text{Integer} \]

\[ k \; x = 3 \times x + f \; x \]
\[ h :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]

\[ h \ x \ y = \ g \ y + f \ (g \ y) \]
where
\[ g \ z = z \times z \]

\[ g :: \text{Integer} \rightarrow \text{Integer} \]
\[ g \ x = 3 \times x + f \ x \]

\[ h' :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]
\[ h' \ x \ y = k \ y + f \ (k \ y) \]
where
\[ g \ z = z \times z \]

\[ k :: \text{Integer} \rightarrow \text{Integer} \]
\[ k \ x = 3 \times x + f \ x \]

\[ f = \text{uninterpret "f"} \]

\[ \text{propertyk} = \text{prove $(x::SInteger) \rightarrow g \ x .== k \ x$} \]
\[ \text{propertyh} = \text{prove $(x::SInteger) \ (y::SInteger) \rightarrow h \ x \ y .== h' \ x \ y$} \]
\[ h :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]

\[ h \ x \ y = g \ y + f \ (g \ y) \]

where

\[ g \ z = z^2 \]

\[ g :: \text{Integer} \rightarrow \text{Integer} \]

\[ g \ x = 3 \times x + f \ x \]

\[ h' :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]

\[ h' \ x \ y = k \ y + f \ (k \ y) \]

where

\[ g \ z = z^2 \]

\[ k :: \text{Integer} \rightarrow \text{Integer} \]

\[ k \ x = 3 \times x + f \ x \]

\[ f = \text{uninterpret} \ "f" \]

\[ \text{property}\_k = \text{prove} \ \\lambda (x::\text{SInteger}) \rightarrow g \ x \ .== \ k \ x \]

\[ \text{property}\_h = \text{prove} \ \\lambda (x::\text{SInteger}) \ (y::\text{SInteger}) \rightarrow h \ x \ y \ .== \ h' \ x \ y \]

*Refac2* \> propertyk

Q.E.D.

*Refac2* \> propertyh

Falsifiable. Counter-example:

\[ s0 = 0 :: \text{SInteger} \]

\[ s1 = -1 :: \text{SInteger} \]
Automatically verify instances of refactorings

Feasible … and open.

Compare with the task of general proofs, which requires …

… semantics and meta-theory for a real language

Can we extract evidence in the positive case, too?
Guaranteeing API and DSL?

API provides a general transformation framework …

… is there any way of ensuring that it can be restricted to support only correct transformations?

Even if not, users can write properties encapsulating the change …

… system can generate proof obligations for the functions and modules affected (SCC and SCCs that use changed functions).

DSL - correctness is ensured by correctness of component refactorings.
Is the approach functional or general?

Extended repertoire of expression-level refactorings.
These are local, and should be amenable to automated verification.

Structural refactorings similar for OO and other examples.

Proof of structural properties made easier by lack of side-effects.
Thank you

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