Refactoring for Functional Programs

Simon Thompson, University of Kent
What have we learned about tool building?

Simon Thompson, University of Kent
Science

Engineering

Human factors
Science
Engineering
Human factors

Usability & Trust
Automation
Languages
What do you mean by “refactoring”? 
solveLEIntAux :: Eq a => Eq b => [[[Rational]], [a], [b]] -> Maybe [(b, Integer)]

solveLEIntAux [] = Nothing

solveLEIntAux (h:t) =
  case splitOrConvert h rSol of
  Just (Left nh) -> solveLEIntAux (nub (t ++ nh))
  Just (Right s) -> Just s

  Nothing -> Nothing

  Nothing -> solveLEIntAux t

  where
  rSol = solveLE h
What does “refactoring” mean?

Minor edits or wholesale changes

Something local or of global scope

Just a general change in the software …

… or something that changes its structure, but not its functionality?

Something chosen by a programmer …

… or chosen by an algorithm?
Expression-level refactorings

HLint Manual

HLint is a tool for suggesting possible improvements to Haskell code. These suggestions include ideas such as using alternative functions, simplifying code and spotting redundancies. This document is structured as follows:

1. Installing and running HLint
2. FAQ
3. Customizing the hints

Acknowledgements

This program has only been made possible by the presence of the haskell-src-exts package, and many improvements have been made by Niklas Broberg in response to feature requests. Additionally, many people have provided help and patches, including Lennart Augustsson, Malcolm Wallace, Henk-Jan van Tuyl, Gwern Branwen, Alex Ott, Andy Stewart, Roman Leshchinskiy and others.

Cleaning up Erlang Code is a Dirty Job
but Somebody’s Gotta Do It

Thanassis Avgerinos
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
ethan@softlab.ntua.gr

Konstantinos Sagonas
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
kostis@cs.ntua.gr
**Expression-level refactorings**

**HLint Manual**

HLint is a tool for suggesting possible improvements to Haskell code. These suggestions include ideas such as using alternative functions, simplifying code and spotting redundancies. This document is structured as follows:

1. Installing and running HLint
2. FAQ
3. Customizing the hints

**Acknowledgements**

This program has only been made possible by the presence of the haskell-src-exts package, and many improvements have been made by Niklas Broberg in response to feature requests. Additionally, many people have provided help and patches, including Lennart Augustsson, Malcolm Wallace, Henk-Jan van Tuyl, Gwern Branwen, Alex Ott, Andy Stewart, Roman Leshchinsky and others.

Sample.hs:5:7: Warning: Use and Found

```
foldr1 (&&)
```

Why not and

Note: removes error on []
What sort of refactoring interests us?

Changes beyond the purely local, which can be effected easily.
What sort of refactoring interests us?

Changes beyond the purely local, which can be effected easily.

Renaming a function / module / type / structure.

Changing a naming scheme: camel_case to camelCase,…

Generalising a function … extracting a definition.
Function extraction in Erlang

Extension and reuse

\[
\text{loop}_a() \rightarrow \\
\hspace{1em} \text{receive} \\
\hspace{2em} \text{stop} \rightarrow \text{ok}; \\
\hspace{2em} \{\text{msg, } \_\text{Msg}, \ 0\} \rightarrow \text{loop}_a(); \\
\hspace{2em} \{\text{msg, Msg, N}\} \rightarrow \\
\hspace{3em} \text{io:format("ping!\n"),} \\
\hspace{3em} \text{timer:sleep(500),} \\
\hspace{3em} b ! \{\text{msg, Msg, N - 1}\}, \\
\hspace{3em} \text{loop}_a() \\
\hspace{1em}\text{end.}
\]
**Function extraction in Erlang**

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.
```

Let’s turn this into a function
Function extraction in Erlang

Extension and reuse

```erlang
code(module, 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.
end).

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.
```
Function extraction in Erlang

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.
end.
```

```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}.
```
What sort of refactoring interests us?

Changes beyond the purely local, which can be effected easily.

Renaming a function / module / type / structure.

Changing a naming scheme: camel_case to camelCase, ...

Generalising a function … extracting a definition.

Changing a type representation.

Changing a library API.

Module restructuring: e.g. removing inclusion loops.
Refactoring tools
Refactoring
=
Transformation
Refactoring

= 

Transformation
Refactoring

= 

Transformation + Pre-condition
How to refactor?

By hand … using an editor

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

Tool-supported

     Handles transformation and analysis.
     Scalable to large-code bases: module-aware.
     Integrated with tests, macros, ...
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).
foo(X) -> io:format(X).
```erlang
-module(foo).
-export([[foo/1, foo/0]]).
foo() -> spawn(foo, foo, [foo]).
foo(X) -> io:format(X).
```
-module(foo).
-export([foo/1,foo/0]).
foo() -> spawn(foo,foo,[foo]).
foo(X) -> io:format(X).
```erlang
-module(foo).
-export([foo/1, foo/0]).
foo() -> spawn(foo, foo, [foo]).
foo(X) -> io:format(X).
```
Traversals, strategies and visitors

Multi-purpose

Collect and analyse info.
Effect a transformation.

Separation of concerns

Point-wise operation …
… and tree traversal
Traversals, strategies and visitors

Multi-purpose

Collect and analyse info.
Effect a transformation.

Separation of concerns

Point-wise operation …
… and tree traversal
Traversals, strategies and visitors

Multi-purpose
- Collect and analyse info.
- Effect a transformation.

Separation of concerns
- Point-wise operation …
- … and tree traversal
Traversals, strategies and visitors

Multi-purpose

Collect and analyse info.
Effect a transformation.

Separation of concerns

Point-wise operation …
… and tree traversal
Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive
<table>
<thead>
<tr>
<th>Haskell</th>
<th>Erlang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly typed</td>
<td>Weakly typed</td>
</tr>
<tr>
<td>Lazy</td>
<td>Strict</td>
</tr>
<tr>
<td>Pure + Monads</td>
<td>Some side-effects</td>
</tr>
<tr>
<td>Complex type system</td>
<td>Concurrency</td>
</tr>
<tr>
<td>Layout sensitive</td>
<td>Macros and idioms</td>
</tr>
<tr>
<td>Language</td>
<td>Features</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Haskell</td>
<td>Strongly typed, Lazy, Pure + Monads, Complex type system, Layout sensitive</td>
</tr>
<tr>
<td>Erlang</td>
<td>Weakly typed, Strict, Some side-effects, Concurrency, Macros and idioms</td>
</tr>
<tr>
<td>OCaml</td>
<td>Strongly typed, Strict, Refs etc and i/o, Modules + interfaces, Scoping/modules</td>
</tr>
<tr>
<td><strong>Haskell</strong></td>
<td><strong>Erlang</strong></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Strongly typed</td>
<td>Weakly typed</td>
</tr>
<tr>
<td>Lazy</td>
<td>Strict</td>
</tr>
<tr>
<td>Pure + Monads</td>
<td>Some side-effects</td>
</tr>
<tr>
<td>Complex type system</td>
<td>Concurrency</td>
</tr>
<tr>
<td>Layout sensitive</td>
<td>Macros and idioms</td>
</tr>
</tbody>
</table>

- **HaRe**
  - Haskell 98
  - Programmatica / GHC Haskell API
  - Basic refactorings, clones, type-based, ...
  - Strategic prog
Haskell
Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang
Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

OCaml
Strongly typed
Strict
Refs etc and i/o.
Modules + interfaces
Scoping/modules

HaRe
Haskell 98
Programmatica /
GHC Haskell API
Basic refactorings,
clones, type-based, …
Strategic prog

Wrangler
Full Erlang
Erlang, syntax_tools
HaRe + module,
API, DSL, context.
Naive strategic prog
<table>
<thead>
<tr>
<th>Language</th>
<th>Type System</th>
<th>Evaluation</th>
<th>Concurrency</th>
<th>Other Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>Strongly typed</td>
<td>Lazy</td>
<td>Pure + Monads</td>
<td>Complex type system, Layout sensitive</td>
</tr>
<tr>
<td>Erlang</td>
<td>Weakly typed</td>
<td>Strict</td>
<td>Concurrency</td>
<td>Macros and idioms</td>
</tr>
<tr>
<td>OCaml</td>
<td>Strongly typed</td>
<td>Strict</td>
<td>Refs etc and i/o.</td>
<td>Modules + interfaces, Scoping/modules</td>
</tr>
<tr>
<td>HaRe</td>
<td>Haskell 98</td>
<td>Programmatica / GHC Haskell API</td>
<td>Basic refactorings, clones, type-based, …</td>
<td>Strategic prog</td>
</tr>
<tr>
<td>Wrangler</td>
<td>Full Erlang</td>
<td>Erlang, syntax_tools</td>
<td>HaRe + module, API, DSL, context.</td>
<td>Naive strategic prog</td>
</tr>
<tr>
<td>ROTOR</td>
<td>(O)Caml compiler</td>
<td>OCaml compiler</td>
<td>So far: renaming &amp; dependency theory.</td>
<td>Derived visitors</td>
</tr>
</tbody>
</table>
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.
Wrangler

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

Clone detection and removal

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

Clone detection and removal

Module structure improvement

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

Clone detection and removal

Module structure improvement

API: define new refactorings

Basic refactorings: structural, macro, process and test-framework related
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
-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.
Analyses needed …

Static semantics

Types

Modules

Side-effects
Analyses needed …

<table>
<thead>
<tr>
<th>Static semantics</th>
<th>Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Process structure</td>
</tr>
<tr>
<td>Modules</td>
<td>Macros</td>
</tr>
<tr>
<td>Side-effects</td>
<td>Conventions and frameworks</td>
</tr>
</tbody>
</table>
Feasible

Desirable

Sustainable
Desirable

Sustainable

Feasible
Renaming
What is in a name?

Resolving names requires not just the static structure …

… but also types (polymorphism, overloading) and modules.

Beyond the wits of regexps.

Leverage other infrastructure or the compiler.
Types sneak in …

\[ f(x) = (x^2 + 42) + (x + 42) \]

\[ f(x, y) = (x^2 + y) + (x + y) \]
Types sneak in …

\[ f \ x = (x^2 + 42) + (x + 42) \]
\[ f \ x \ y = (x^2 + y) + (x + y) \]

funny = length (\[\text{[True]}\] ++ []) + length (\[\text{[True]}\] ++ [])

funny \ xs = length (\[\text{[True]}\] ++ \xs) + length (\[\text{[True]}\] ++ \xs)
... as do different sorts of atoms

```prolog
-module(foo).
-export([foo/1, foo/0]).

foo() -> spawn(foo, foo, [foo]).

foo(X) -> io:format("~w", [X]).
```
And some peculiarities

\[ f1(P) \rightarrow \]

\texttt{receive}

\{ok, X\} \rightarrow P!thanks;

\{error,\_\} \rightarrow P!grr

end,

\[ P!\{\text{value},X\}. \]
And some peculiarities

\[
f_1(P) \rightarrow \\
\quad \text{receive} \\
\quad \quad \{\text{ok, } X\} \rightarrow P!\text{thanks}; \\
\quad \quad \{\text{error, } \_\} \rightarrow P!\text{grr} \\
\quad \text{end,} \\
\quad P!\{\text{value, } X\}.
\]

\[
f_2(P) \rightarrow \\
\quad \text{receive} \\
\quad \quad \{\text{ok, } X\} \rightarrow P!\text{thanks}; \\
\quad \quad \{\text{error, } X\} \rightarrow P!\text{grr} \\
\quad \text{end,} \\
\quad P!\{\text{value, } X\}.
\]
Abandon any idea of building language-independent refactoring tools.
OCaml’s module system
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end
OCaml modules

```ocaml
module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end
```
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=", 1))) ;;
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int)(Pair(String)(Int));;
print_endline (P.to_string (0, ("!=", 1)));;
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int)(Pair(String)(Int));;
print_endline (P.to_string (∅, ("!=", 1)));;
OCaml modules

```ocaml
module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int)(Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=" , 1))) ;;
```
OCaml modules

```ocaml
module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int) (Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!=" , 1))) ;;
```
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int) (Pair(String)(Int)) ;;
print_endline (P.to_string (0, ("!="", 1))) ;;
OCaml modules

module type Stringable = sig
  type t
  val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
  type t = X.t * Y.t
  let to_string (x, y) =
    (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
  type t = int
  let to_string i = int_to_string i
end

module String = struct
  type t = string
  let to_string s = s
end

module P = Pair(Int)(Pair(String)(Int));;
print_endline (P.to_string (0, ("!="), 1));;
OCaml modules

```ocaml
module type Stringable = sig
    type t
    val to_string : t -> string
end

module Pair(X : Stringable)(Y : Stringable) = struct
    type t = X.t * Y.t
    let to_string (x, y) =
        (X.to_string x) ^ " " ^ (Y.to_string y)
end

module Int = struct
    type t = int
    let to_string i = int_to_string i
end

module String = struct
    type t = string
    let to_string s = s
end

module P = Pair(Int)(Pair(String)(Int));;
print_endline (P.to_string (0, ("!=", 1)));;
```
Formalised using Coq.

Characterising renaming by value extension kernels.

Abstract renaming semantics, proved adequate:

"Two equal abstractions have equal concrete versions"

Characterising Renaming within OCaml’s Module System: Theory and Implementation

Reuben N. S. Rowe
Hugo Férée
Simon J. Thompson
Scott Owens
School of Computing, University of Kent, Canterbury, UK
{r.n.s.rowe,h.feree,s.j.thompson,s.a.owens}@kent.ac.uk

Abstract

We present an abstract, set-theoretic denotational semantics for a significant subset of OCaml and its module system, allowing us to reason about the correctness of renaming value bindings. Our semantics captures information about the binding structure of programs, as well as about which declarations are related by the use of different language constructs (e.g., functions, module types and module constraints). Correct reimaginings are precisely those that preserve this structure. We show that our abstract semantics is sound with respect to a domain-theoretic denotational model of the operational behaviour of programs, and that it allows us to prove various high-level, intuitive properties of reimaginings. This formal framework has been implemented in a prototype refactoring tool for OCaml that performs renaming.

CCS Concepts - Theory of computation → Abstraction,
Denotational semantics; Program constructs; Functional constructs; Software and its engineering → Software maintenance tools.

Keywords Adequacy, dependencies, modules, module types, OCaml, refactoring, renaming, semantics.

ACM Reference Format:

ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314600

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission is granted to the Foundation for Open Access Publishing (FOAP) to include this article in FOAP's aggregate archive, and to use FOAP's digital services to distribute this article in the FOAP aggregated archive.

11 Introduction

Refactoring is the process of changing how a program works without changing what it does, and is a necessary and ongoing process in both the development and maintenance of any codebase [22]. Whilst individual refactoring steps are often conceptually very simple, applying them in practice can be complex, involving many repeated but subtly varying changes across the entire codebase. Moreover refactorings are, by and large, context sensitive, meaning that carrying them out by hand can be error-prone and the use of general-purpose utilities (even powerful ones such as grep and sed) is only effective up to a point.

This immediately poses a challenge, but also presents an opportunity. The challenge is how to ensure, or check, a proposed refactoring does not change the behaviour of the program (or does so only in very specific ways). The opportunity is that since refactoring is fundamentally a mechanistic process it is possible to automate it. Indeed, it is desirable in order to avoid human-introduced errors. Our aim in this paper is to outline how we might begin to provide a solution to the dual problem of specifying and verifying the correctness of refactorings and building correct-by-construction automated refactoring tools for OCaml [22, 31].

Renaming is a quintessential refactoring, and so it is on this that we focus as a first step. Specifically, we look at renaming the bindings of values in modules. One might very well be tempted to claim that, since we are in a functional setting, this is simply α-conversion (as in λ-calculi) and thus trivial. This is emphatically not the case. OCaml utilises language constructs, particularly in its module system, that behave in fundamentally different ways to traditional variable binders. Thus, to carry out renaming in OCaml correctly, one must take the meaning of these constructs into account.

Some of the issues are illustrated by the example program in fig. 1 below. This program defines a constructor Pair taking two modules as arguments, which must conform to the Stringable module type. It also defines two structures Int and String. It then uses these as arguments in applications of Pair, the result of which is bound as the module P. To rename the to_string function in the module Int correctly, we must take the following into account.
Building tools can lead us to re-think theory.
Clone detection
Duplicate code considered harmful

It’s a bad smell …

- increases chance of bug propagation,
- increases size of the code,
- increases compile time, and,
- increases the cost of maintenance.

But … it’s not always a problem.
What is similar code?

\[(x+3)+4\] \[4+(5-(3x))\]
What is similar code?

\[ (X+3) + 4 \]

\[ 4 + (5 - (3 \times X)) \]
What is similar code?

The anti-unification gives the (most specific) common generalisation.
What is similar code?

\[ (X+3) + 4 \]
\[ 4 + (5 - (3 \times X)) \]
\[ X + Y \]

The anti-unification gives the (most specific) common generalisation.
What is similar code?

The anti-unification gives the (most specific) common generalisation.
What makes a clone (in Erlang)?

Thresholds

Number of expressions

Number of tokens

Number of variables introduced

Similarity = \( \min_{i=1...n}(\frac{\text{size}(\text{Gen})}{\text{size}(E_i)}) \)
What makes a clone (in Erlang)?

Thresholds … and their defaults

- Number of expressions $\geq 5$
- Number of tokens $\geq 20$
- Number of variables introduced $\leq 4$
- Similarity $= \min_{i=1..n}(\text{size}(\text{Gen})/\text{size}(E_i)) \geq 0.8$
Clone detection and removal

Find a clone, name it and its parameters, and eliminate.

What could go wrong?
What could go wrong?

Naming can’t be automated, nor the order of eliminating.

Bottom-up or top-down?

Widows and orphans, sub-clones, premature generalisation, …
What could go wrong?

new_fun(FilterName, NewVar_1) ->
    FilterKey = ?SMM_CREATE_FILTER_CHECK(FilterName),
    %%%Add rulesets to filter
    RuleSetNameA = "a",
    RuleSetNameB = "b",
    RuleSetNameC = "c",
    RuleSetNameD = "d",
    %%%Remove rulesets
    NewVar_1,
    {RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD, FilterKey}.

Widows and orphans, sub-clones, premature generalisation, ...

new_fun(FilterName, FilterKey) ->
    %%%Add rulesets to filter
    RuleSetNameA = "a",
    RuleSetNameB = "b",
    RuleSetNameC = "c",
    RuleSetNameD = "d",
    ... 16 lines which handle the rules sets are elided ...
    %%%Remove rulesets
    {RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD}. 
What could go wrong?

Naming can’t be automated, nor the order of eliminating.
Bottom-up or top-down?
Widows and orphans, sub-clones, premature generalisation, …
Bring in the experts

With a domain expert …

   can choose in the right order,

   name the clones and their parameters, …

And the domain expert can learn in the process …

   e.g. test code example from Ericsson.
Support user involvement rather than full automation.
Desirable

Feasible

Sustainable
Obstacles

Observations

Incentives
Obstacles

Incentives

Observations
<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Wrangler</th>
<th>LambdaStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fold against macro</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Fold expression against function</td>
<td>84</td>
<td>17</td>
</tr>
<tr>
<td>Generalisation</td>
<td>46</td>
<td>8</td>
</tr>
<tr>
<td>Inline variable</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Introduce new variable</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>Move function between modules</td>
<td>229</td>
<td>14</td>
</tr>
<tr>
<td>Function extraction</td>
<td>119</td>
<td>87</td>
</tr>
<tr>
<td>Introduce new macro</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Rename function</td>
<td>236</td>
<td>19</td>
</tr>
<tr>
<td>Rename module</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>Rename variable</td>
<td>425</td>
<td>6</td>
</tr>
<tr>
<td>Introduce tuple</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Unfold function application</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Modularity inspection</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Keep it simple!
User observations

Comprehension exercise on student coursework.

Clone detection exercise with Ericsson staff.

Workflow integration at LambdaStream.

Developing and using DSL with Quviq.

Sitting-in with OCaml group at Jane Street.
Why not?

We can do things it would take too long to do without a tool.

We can be less risk-averse: e.g. in doing generalisation.

Exploratory: try and undo if we wish.

95% ≫ 0%: hit most cases … fix the last 5% “by hand”. 
Concrete incentives

Quviq

Routine task of removing code instrumentation before shipping.

Estimated 1 person-month of savings per annum.

Jane Street

Compliance overhead

Reduce the cost of code review for refactorings like renamings …

… if a tool is trusted.
The ecosystem

Editor integration … but which are the most popular?

LSP support.

Build and test tools, pre-processors.

Dependencies … and Windows.
Benefits should outweigh costs.
Appearance must be right

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

my_funny_list() ->
[ foo,
  ,bar,
  ,baz,
  ,wombat
]
Appearance must be right

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

{v1, v2, v3}

{v1,v2,v3}

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

{v1,v2,v3}
Appearance must be right

```erlang
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
```
Appearance must be right

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

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

{v1, v2, v3}
{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 appearance

Preserve precisely parts not touched.

Pretty print … or use lexical details.
Preserving appearance isn’t built in

Compilers throw away some / all layout info, comments, …

Need to build infrastructure to hide layout manipulations.

Learn layout for synthesised code from existing codebase?

*Scrap Your Reprinter* by Orchard et al
Yaron Minsky @yminsky · 3h
Just flipped a big codebase over to doing automatic formatting (indentation, line-breaking, whether to put ;;‘s after a toplevel declaration, etc). There are some regressions in readability, but there is something freeing about it. Nothing like not needing to make choices...

Don Stewart @donsbot · 3h
We have data showing how much faster code review is when format is removed from the equation. It's a clear win at scale.
“but there is something freeing about it. Nothing like not needing to make choices …”
I have types ... I don’t need a tool
Up to 90% of refactorings done by hand
SOFTWARE PROJECT MAINTENANCE IS WHERE HASKELL SHINES.

https://www.fpcomplete.com/blog/2016/12/software-project-maintenance-is-where-haskell-shines

As someone unfamiliar with the codebase I wanted to make major changes to the GHC abstract syntax tree, to support API Annotations.

GHC is a big codebase.

I found that it was a straightforward process to change the data type and then fix the compilation errors. Even in the dark bowels of the beast, such as the typechecker.

I think the style of the codebase helps a lot in this case, with lots of explicit pattern matching so that it is immediately obvious when something needs to be changed.

https://www.reddit.com/r/haskell/comments/65d510/experience_reports_on_refactoring_haskell_code/
But is it really as simple as that … ?

Changes in bindings – e.g. name capture – can give code that compiles and type checks, but gives different results.

Are you really prepared to fix 1,000 type error messages?

Maybe just be risk averse …
Ian Jeffries @light_industry · Jan 28
Very bad Haskell code can be worse than bad Python code (if it does pretty much everything in IO and uses very general types like HashMap Text Text everywhere), but this hopefully isn't super common.

Andreas Källberg @Anka213 · Jan 29
Haskell is also very easy and safe to refactor. So even if you have a very bad code-base, you could fairly mechanically and safely transform it until you have better code.

For example, you could newtype a specific case and then update functions until it typechecks.

Alex Nedelcu @alexelcu · Jan 29
I don’t think marketing Haskell as “very easy/safe to refactor” is smart b/c as a matter of fact there are code bases for which this isn’t easy or safe. I hope there are b/c otherwise it means Haskell isn’t used for real world projects and AFAIK that ain’t true.
moduleDef :: LParser Module
moduleDef = do
    reserved "module"
    modName <- identifier
    reserved "where"
    imports <- layout importDef (return [])
    decls <- layout decl (return [])
    cnames <- get
    return $ Module modName imports decls cnames
From **Monad** to **Applicative**

```haskell
moduleDef :: LParser Module
moduleDef = do
    reserved "module"
    modName <- identifier
    reserved "where"
    imports <- layout importDef (return ())
    decls <- layout decl (return ())
    cnames <- get
    return $ Module modName imports decls cnames
```

```haskell
moduleDef :: LParser Module
moduleDef = Module
    <$> (reserved "module" <*> identifier <*> reserved "where")
    <$> layout importDef (return ())
    <$> layout decl (return ())
    <$> get
```
From **List** to **Vector**

- `map :: (a -> b) -> [a] -> [b]`
- `app :: [a] -> [a] -> [a]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `take :: Int -> [a] -> [a]`
From **List** to **Vector**

```
map :: (a -> b) -> [a] -> [b]
app :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
take :: Int -> [a] -> [a]
```

```
vmap :: (a -> b) -> (Vec n a) -> (Vec n b)
vapp :: (Vec n a) -> (Vec m a) -> (Vec n+m a)
vfilter :: (a -> Bool) -> (Vec n a) -> (Vecs n a)
vtake :: (n :: Int) -> (Vec m a) -> (Vec (min n m) a)
vtake :: (n :: Int) -> (Vec m a) -> (Vecs n a)
```
Types vs refactoring?

The more precise the typings, the more fragile the structure.

Difficulty of getting it right first time: Vec vs Vecs vs ...

vmap :: (a -> b) -> (Vec n a) -> (Vec n b)
vapp :: (Vec n a) -> (Vec m a) -> (Vec n+m a)
vfilter :: (a -> Bool) -> (Vec n a) -> (Vecs n a)
vtake :: (n :: Int) -> (Vec m a) -> (Vec (min n m) a)
vtake :: (n :: Int) -> (Vec m a) -> (Vecs n a)
Types can both help and hinder effective refactoring.
Why should I trust your refactoring tool on my code?
Refactoring Tools Are Trustworthy Enough

John Brandt

Refactoring tools don’t have to guarantee correctness to be useful. Sometimes imperfect tools can be particularly helpful.

A COMMON DEFINITION of refactoring is “a behavior-preserving transformation that improves the overall code quality.” Code quality is subjective, and a particular refactoring in a sequence of refactorings often might temporarily make the code worse. So, the code-quality-improvement part of the definition is often omitted, which leaves that refactorings are simply behavior-preserving transformations.

From that definition, the most important part of tool-supported refactorings appears to be correctness in behavior preservation. However, from a developer’s viewpoint, the most important part is the refactoring’s usefulness: can it help developers get their job done better and faster? Although absolute correctness is a necessary condition for developers to use an automated refactoring tool.

Consider an imperfect refactoring tool. If a developer needs to perform a refactoring that the tool provides, he or she has two options. The developer can either use the tool and fix the bugs it introduced or perform manual refactoring and fix the bugs the manual changes introduced. If the time spent using the tool and fixing the bugs is less than the time doing it manually, the tool is useful. Furthermore, if the tool supports preview and undo, it can be more useful. With previewing, the developer can double-check that the changes look correct before they’re saved; with undo, the developer can quickly reverse the changes if they introduced any bugs.

Often, even a buggy refactoring tool is more useful than an automated refactoring tool that never introduces bugs. For example, automated tools often can’t check all the preconditions for a refactoring. The preconditions might be undecidable, or no efficient algorithm exists for checking them. In this case, the buggy tool might check as much as it can and proceed with the refactoring, whereas the correct version sees that it can’t check everything it needs and aborts the refactoring, leaving the developer to perform it manually. Depending on the buggy tool’s defect rate and the developer’s abilities, the buggy tool might introduce fewer errors than the correct tool paired with manual refactoring.

Even when a refactoring can be implemented without bugs, it can be beneficial to relax some preconditions to allow non-behavior-preserving transformations. For example, after implementing Extract Method in the Smalltalk Refactoring Browser, my colleagues and I received an email requesting that we allow the extracted method to override minor or even larger behavior changes go unnoticed, are tolerated, or are even welcomed (because refactoring the code has revealed logical errors). I assume that this conception of refactoring is by far the most common, and I have no objections to it (other than, perhaps, that I would question such a software process per se).

Now imagine a scenario in which the bugs aren’t troubling enough to force the tool to fix them. This of course has the corollary that the bugs aren’t troubling enough to be fixed (because otherwise, the necessary resources would be made available). For this corollary, two explanations are common: “Hardly anyone uses refactoring tools anyway, so who cares about the bugs?” and “The bugs aren’t a real problem; my compiler and test suite will catch them as I go.” I reject both explanations.

A developer can quickly revert the changes if they introduced any bugs. With undo, the developer can quickly reverse the changes if they introduced any bugs. Refactoring tools don’t have to guarantee correctness to be useful. Sometimes imperfect tools can be particularly helpful.

Trust Must Be Earned

Friedrich Steimann

Creating bug-free refactoring tools is a real challenge. However, tool developers will have to meet this challenge for their tools to be truly accepted.

WHEN I ASK people about the progress of their programming projects, I often get answers like “I got it to work—now I need to do some refactoring!” What they mean is that they managed to tweak their code so that it appears to do what it’s supposed to do, but knowing the process, they realize all too well that its result won’t pass even the lightest code review. In the following refactoring phase, whether it’s manual or tool supported, minor or even larger behavior changes go unnoticed, are tolerated, or are even welcomed (because refactoring the code has revealed logical errors). I assume that this conception of refactoring is by far the most common, and I have no objections to it (other than, perhaps, that I would question such a software process per se).

Now imagine a scenario in which the code has undergone extensive (and expensive) certification. If this code is touched in multiple locations, chances are that the entire certification must be repeated. Pervasive changes typically become necessary if the functional requirements change and the code’s current design can’t accommodate the new requirements in a form that would allow isolated certification of the changed code. If, however, we had refactoring tools that have been certified to preserve behavior, we might be able to refactor the code so that the necessary functional changes remain local and don’t require global recertification of the software. Unfortunately, we don’t have such tools.

What we have today is the common sentiment that “if only the tool people had enough resources, they would fix the refactoring bugs,” suggesting that no fundamental obstacles to fixing them exist. This of course has the corollary that the bugs aren’t troubling enough to be fixed (because otherwise, the necessary resources would be made available). For this corollary, two explanations are common: “Hardly anyone uses refactoring tools anyway, so who cares about the bugs?” and “The bugs aren’t a real problem; my compiler and test suite will catch them as I go.” I reject both explanations.
Challenges to and Solutions for Refactoring Adoption
An Industrial Perspective

Tushar Sharma and Girish Buryanarayana, Siemens Technology and Services Private Limited
Ganesh Samarthyam, independent consultant and corporate trainer

Several practical challenges must be overcome to facilitate industry’s adoption of refactoring. Results from a Siemens Corporate Development Center India survey highlight common challenges to refactoring adoption. The development center is devising and implementing ways to meet these challenges.

Software suffering from technical debt requires significant effort to maintain and extend. A key approach to managing technical debt is refactoring. William Opdyke defined refactoring as “behavior-preserving program transformation.” Martin Fowler’s seminal work increased refactoring’s popularity and extended its academic and industrial reach. Modern software development methods such as Extreme Programming (“refactor mercilessly”) have adopted refactoring as an essential element.

However, our experience assessing industrial software design and training software architects and developers at Siemens Corporate Development Center India (CT DC IN) has revealed numerous challenges to refactoring adoption in an industrial context. So, we surveyed CT DC IN software architects to understand these challenges. Although we knew many of the problems facing refactoring adoption, our survey gave us insight into how these challenges ranked within CT DC IN. Drawing on this insight, we outline solutions to the challenges and briefly describe key CT DC IN initiatives to encourage refactoring adoption. We hope our survey findings and refactoring-centric initiatives help move the software industry toward wider, more effective refactoring adoption.

Survey Details
CT DC IN is a core software development center for Siemens products. Its software systems pertain to different Siemens sectors (Industry, Healthcare, Infrastructure & Cities, and Energy); address diverse domains, are built on different platforms, and are in various development and maintenance stages. CT DC IN, which has increasingly focused on improving its software’s internal quality, wanted to understand the organization’s status quo regarding technical debt, code and design smells, and refactoring. Furthermore, recent internal design assessments and training sessions revealed challenges to refactoring adoption. To better understand these deterrents—and thereby adopt appropriate measures to address them—we conducted our survey.

Breaking code
Cannot justify the time spent
Unpredictable impact
Difficult to review
Inadequate tools
Challenges to and Solutions for Refactoring Adoption
An Industrial Perspective

Tushar Sharma and Girish Buryanarayana, Siemens Technology and Services Private Limited
Ganesh Samarthyam, independent consultant and corporate trainer

Several practical challenges must be overcome to facilitate industry’s adoption of refactoring. Results from a Siemens Corporate Development Center India survey highlight common challenges to refactoring adoption. The development center is devising and implementing ways to meet these challenges.

Industrial software systems typically have complex, evolving code bases that must be maintained for many years. It’s important to ensure that such systems’ design and code don’t decay or accumulate technical debt. Software suffering from technical debt requires significant effort to maintain and extend.

A key approach to managing technical debt is refactoring. William Opdyke defined refactoring as “behavior-preserving program transformation.” Martin Fowler’s seminal work increased refactoring’s popularity and extended its academic and industrial reach. Modern software development methods such as Extreme Programming (“refactor mercilessly”) have adopted refactoring as an essential element.

However, our experience assessing industrial software design and training software architects and developers at Siemens Corporate Development Center India (CT DC IN) has revealed numerous challenges to refactoring adoption in an industrial context. So, we surveyed CT DC IN software architects to understand these challenges. Although we knew many of the problems facing refactoring adoption, our survey gave us insight into how these challenges ranked within CT DC IN. Drawing on this insight, we outline solutions to the challenges and briefly describe key CT DC IN initiatives to encourage refactoring adoption. We hope our survey findings and refactoring-centric initiatives help move the software industry toward wider, more effective refactoring adoption.

Survey Details
CT DC IN is a core software development center for Siemens products. Its software systems pertain to different Siemens sectors (Industry, Healthcare, Infrastructure & Cities, and Energy), address diverse domains, are built on different platforms, and are in various development and maintenance stages.

CT DC IN, which has increasingly focused on improving its software’s internal quality, wanted to understand the organization’s status quo regarding technical debt, code and design smells, and refactoring. Furthermore, recent internal design assessments and training sessions revealed challenges to refactoring adoption. To better understand these deterrents—and thereby adopt appropriate measures to address them—we conducted our survey.

Breaking code
Cannot justify the time spent
Unpredictable impact
Difficult to review
Inadequate tools
Preserving meaning
Do these two programs mean the same thing?

Difficult to examine and compare the meanings directly …

… so we look at other ways of trying to answer this.
Different scopes

- **main module**
- "all" modules
- "all" functions
Different contexts

All tests for the project.

Refactorings need to be test-framework aware

   Naming conventions: foo and foo_test ...

   Macro use, etc.

The makefile for the project.

Using these versions of these libraries … which we don’t control.
Assuring meaning preservation

<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of the refactoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the refactoring itself</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Assuring meaning preservation

<table>
<thead>
<tr>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of</td>
<td>Rename <code>foo</code> to <code>bar</code> in</td>
</tr>
<tr>
<td>the refactoring</td>
<td>this project.</td>
</tr>
<tr>
<td>the refactoring itself</td>
<td></td>
</tr>
</tbody>
</table>
# Assuring meaning preservation

<table>
<thead>
<tr>
<th>instances of the refactoring</th>
<th>Rename <code>foo</code> to <code>bar</code> in this project.</th>
</tr>
</thead>
<tbody>
<tr>
<td>the refactoring itself</td>
<td>Renaming for all names, functions and projects.</td>
</tr>
</tbody>
</table>

| test | verify |

- Verify instances of the refactoring
- Rename `foo` to `bar` in this project.
<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>the refactoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the refactoring itself</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Testing
<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of the refactoring</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>the refactoring itself</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testing new vs old (with Huiqing Li)

Compare the results of `function1` and `function1` (unmodified) …

… using existing unit tests, and randomly-generated inputs

… could compare ASTs as well as behaviour (in former case).
<table>
<thead>
<tr>
<th>instances of the refactoring</th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>the refactoring itself</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Fully random

Generate random modules,

… generate random refactoring commands,

… and check $\equiv$ with random inputs. (w/ Drienyovszky, Horpácsi).
Verification
<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the refactoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the refactoring itself</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Tool verification (with Nik Sultana)

\( \forall p. (Q p) \rightarrow (T p) \simeq p \)

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.
Shallow embedding
<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of the refactoring</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>the refactoring itself</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

DEMUR work with Colin Runciman
\[
\begin{align*}
  h & : \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
  h\ x\ y &= g\ y + f\ (g\ y) \\
  & \quad \text{where} \\
  & \quad g\ z = z*z \\
  g & : \text{Integer} \rightarrow \text{Integer} \\
  g\ x &= 3*x + f\ x \\
  \\
  h' & : \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
  h'\ x\ y &= k\ y + f\ (k\ y) \\
  & \quad \text{where} \\
  & \quad g\ z = z*z \\
  k & : \text{Integer} \rightarrow \text{Integer} \\
  k\ x &= 3*x + f\ x \\
  \\
  f &= \text{uninterpret } "f" \\
  \text{property}\ k &= \text{prove } (x:\text{SInteger}) \Rightarrow g\ x \sim k\ x \\
  \text{property}\ h &= \text{prove } (x:\text{SInteger})\ (y:\text{SInteger}) \Rightarrow h\ x\ y \sim h'\ x\ y
\end{align*}
\]
\[ h :: \text{Integer}\to\text{Integer}\to\text{Integer} \]
\[
h \ x \ y = g \ y + f \ (g \ y) \\
\text{where} \\
g \ z = z \times z \\
g :: \text{Integer}\to\text{Integer} \\
g \ x = 3 \times x + f \ x \\
\\
h' :: \text{Integer}\to\text{Integer}\to\text{Integer} \\
h' \ x \ y = k \ y + f \ (k \ y) \\
\text{where} \\
g \ z = z \times z \\
k :: \text{Integer}\to\text{Integer} \\
k \ x = 3 \times x + f \ x \\
\]

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

property\(k\) = prove \(\forall (x::\text{SInteger}) \rightarrow g \ x .== k \ x\)

property\(h\) = prove \(\forall (x::\text{SInteger}) \ (y::\text{SInteger}) \rightarrow h \ x \ y .== h' \ x \ y\)

*Refac2> property\(k\)
Q.E.D.

*Refac2> property\(h\)
Falsifiable. Counter-example:
  \(s0 = 0 :: \text{SInteger}\)
  \(s1 = -1 :: \text{SInteger}\)
<table>
<thead>
<tr>
<th></th>
<th>test</th>
<th>verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of the refactoring</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>the refactoring itself</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Trust is a complicated, multi-dimensional issue … but we’re working on it.
Re-use don’t reinvent

Compiler front ends are available …

… even if they don’t quite support all we need,

… such as layout preservation, types, …

Keeping up with language evolution, hopefully.

But libraries aren’t necessarily maintained: e.g. Strafunski.
Open Source

Increases trust.

Invites contributors: a shout out to

... Alan Zimmermann, who ported Hare to GHC API,

... Richard Carlsson, who adapted and extended Wrangler,

... and a number of others.

Editor integration: Language Server Protocol will help.
System openness

Open Source … confidence in the code … other committers.

Openness of the system …

… you can check the changes that a refactoring makes,

… and for the DSL can see which refactorings performed
Extensibility
API: templates and rules ... in Erlang

?RULE(Template, NewCode, Cond)

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

```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_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.

---

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

?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 )
]).
It’s better to implement libraries, APIs and DSLs than individual refactorings.
What is the ideal language supporting refactoring?
What’s the ideal language for refactoring?

Changes are first class.

No layout choice: you have to conform to layout rules.

No macros, reflection, …

Compiler stability

Integration with a semantically-aware change management tool.

Theory of patches, …
Feasible

Desirable

Sustainable
Obstacles
Observations
Incentives
https://github.com/alanz/HaRe
https://www.cs.kent.ac.uk/projects/wrangler
https://gitlab.com/trustworthy-refactoring/refactorer