Refactoring and Semantics

Simon Thompson

University of Kent & Eötvös Loránd University

Why should I trust your refactoring tool?

Refactoring

Refactoring tools

Motivation

Trust

Refactoring

Refactoring tools

Motivation

Trust

Engineering

Verification

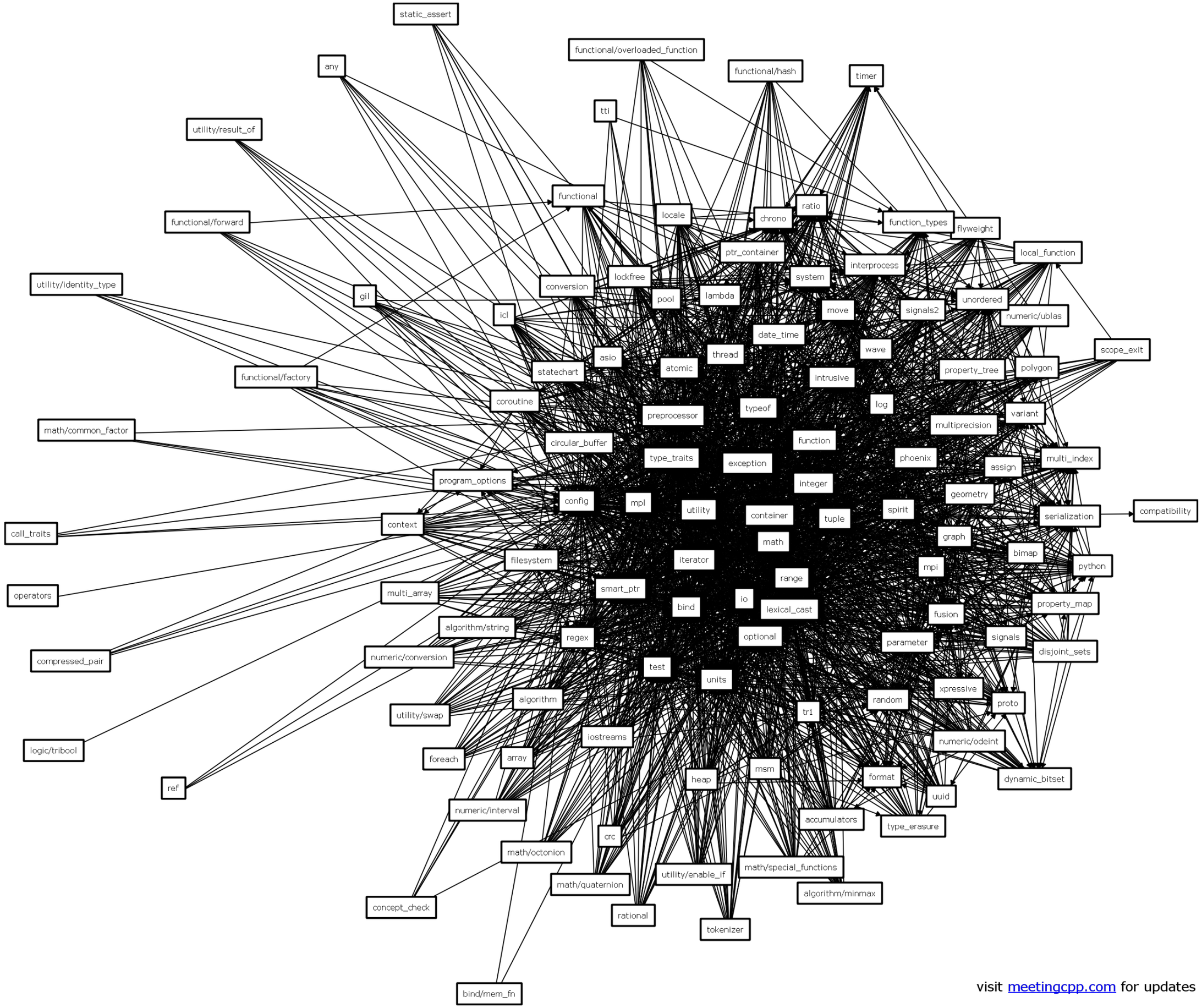
Semantics

What do you mean by “refactoring”?

3 src/EqSolve.hs

View

```
@@ -187,11 +187,12 @@ splitOrConvert (m, r, c) sol =
187 187       Nothing -> Nothing
188 188
189 189  solveLEIntAux :: Eq a => Eq b => [([[Rational]], [a], [b])] -> Maybe [(b, Integer)]
    190 +solveLEIntAux [] = Nothing
190 191  solveLEIntAux (h:t) =
191 192    case splitOrConvert h rSol of
192 193      Just (Left nh) -> solveLEIntAux (nub (t ++ nh))
193 194      Just (Right s) -> Just s
194     -    Nothing -> Nothing
    195 +    Nothing -> solveLEIntAux t
195 196    where
196 197      rSol = solveLE h
197 198
```



visit meetingcpp.com for updates


© Sheila Terry/Science Photo Library

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

by Neil Mitchell

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

by Neil Mitchell

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.

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 done easily.

What sort of refactoring interests us?

Changes beyond the purely local, which can be done 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

```
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

```
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}.
```

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.

Why should I use your refactoring tool?

Refactoring

=

Transformation

Refactoring

=

Transformation + Analysis

How to refactor?

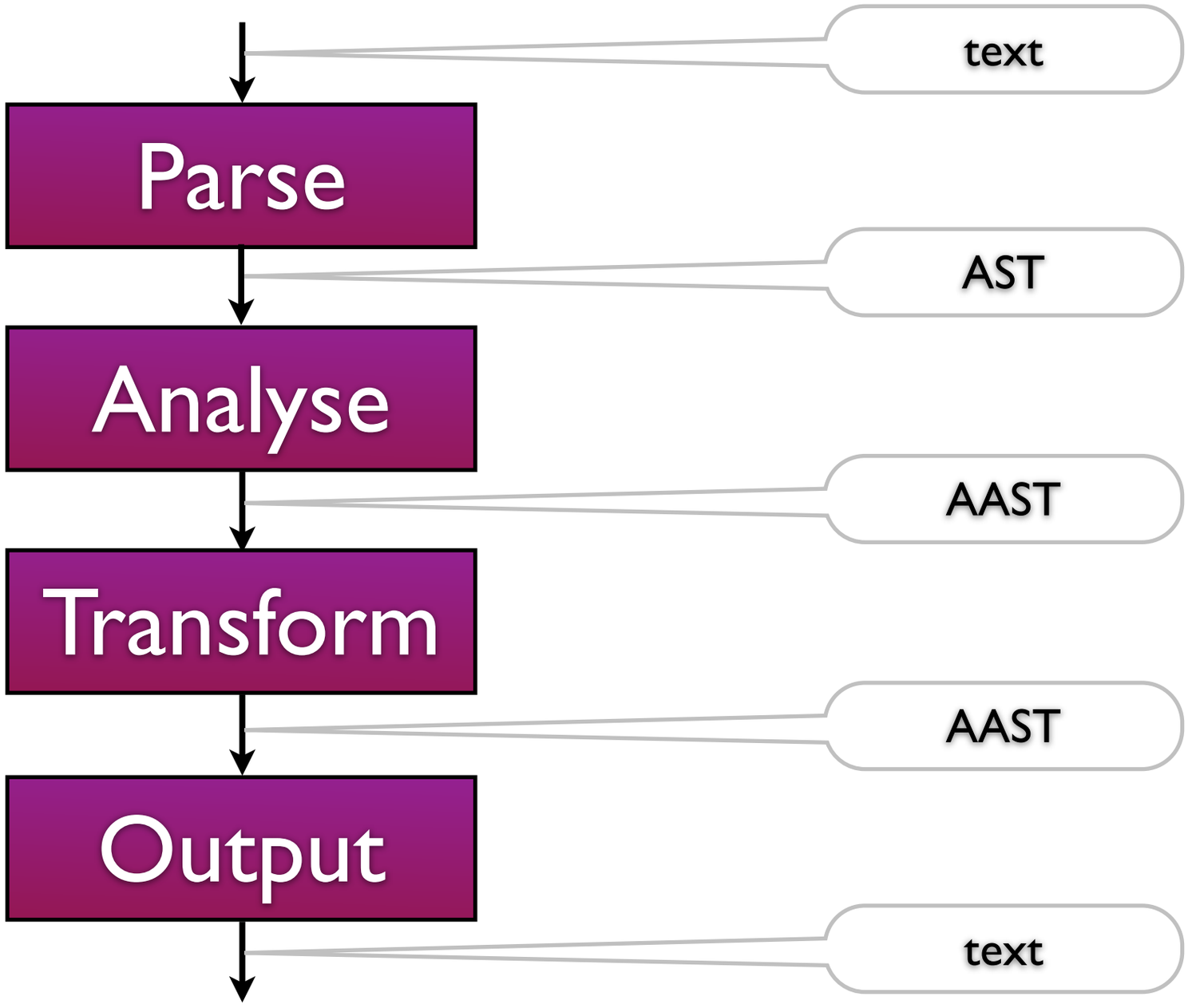
By hand … using an editor

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

Tool-supported

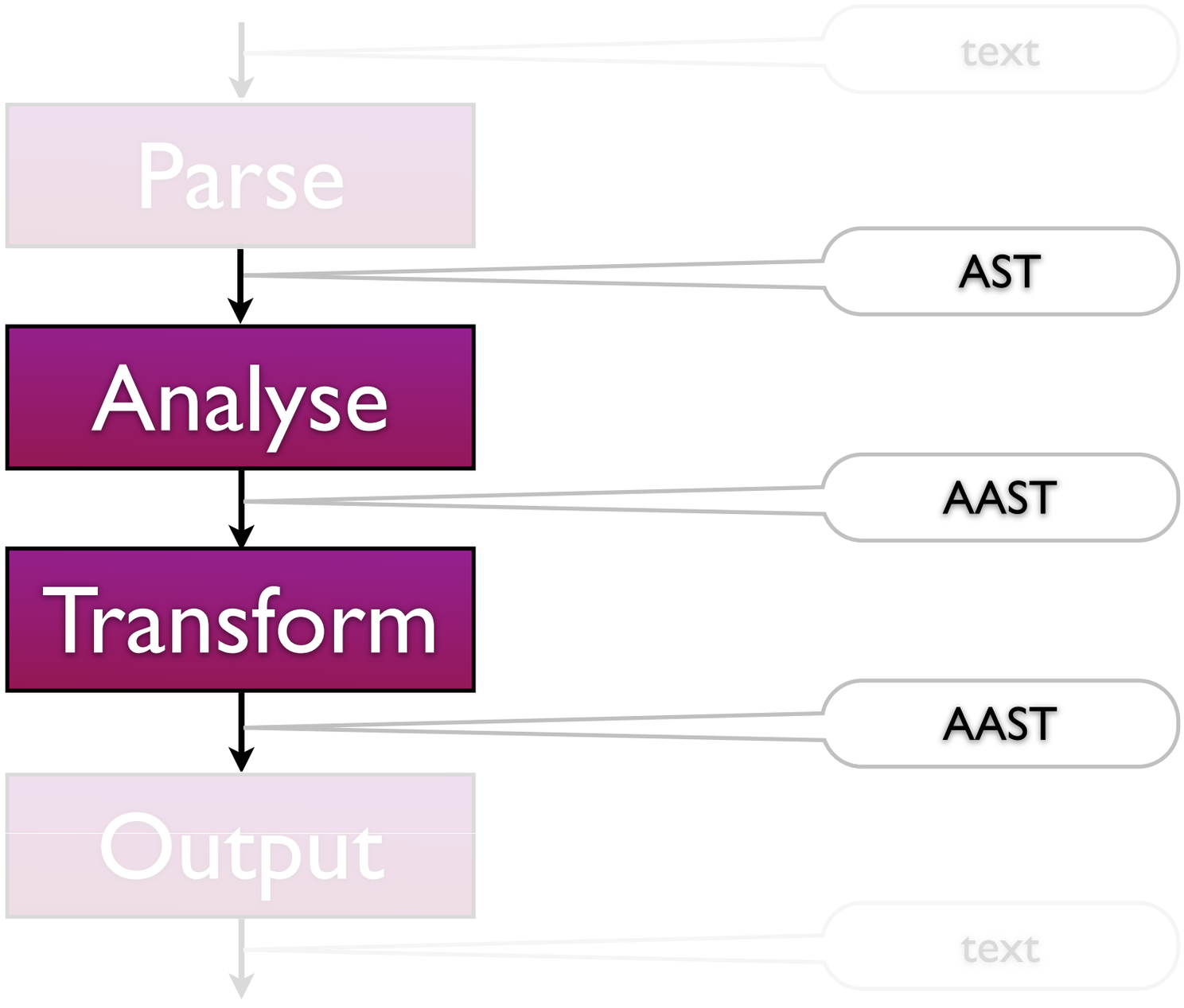
- Handles transformation *and* analysis.
- Scalable to large-code bases: module-aware.
- Integrated with tests, macros, ...

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

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

foo(X) -> io:format(X).
```




```
-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).
```




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

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

foo(X) -> io:format(X).
```

Semantics-based analyses …

Static semantics

Types

Modules

Side-effects

Semantics-based analyses …

Static semantics

Types

Modules

Side-effects

Atoms

Process structure

Macros

Conventions and frameworks

Refactoring and semantics

Core semantics can be straightforward, but …

… there are some dark corners, and

… the language / ecosystem boundary isn't clear.

So, why use a tool?

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.

More adventurous: try and undo if we wish.

95% ≫ 0%: hit most cases … fix the last 5% "by hand".

Wrangler in a nutshell

Automate the simple things, and …

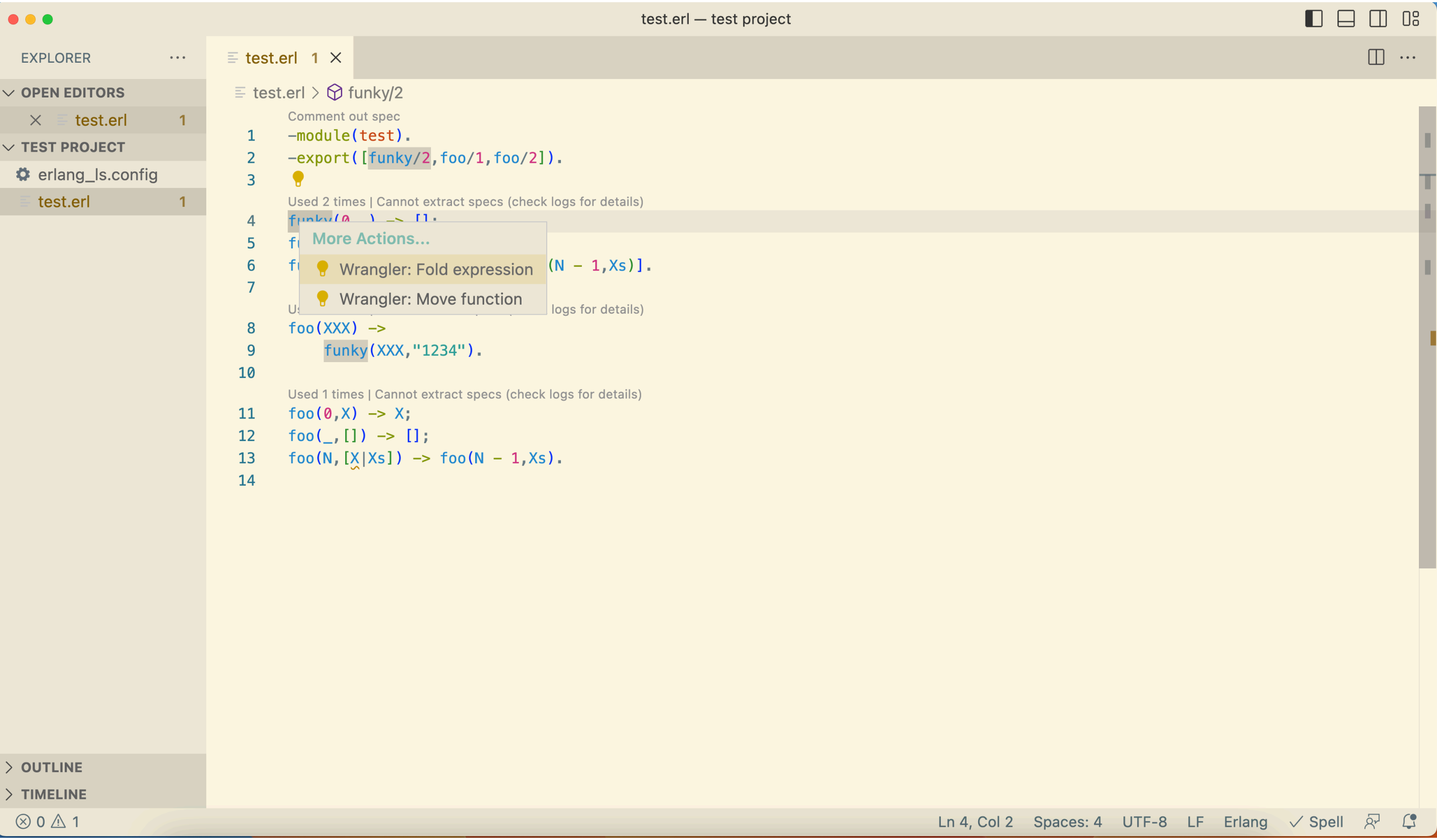
… provide decision support tools otherwise.

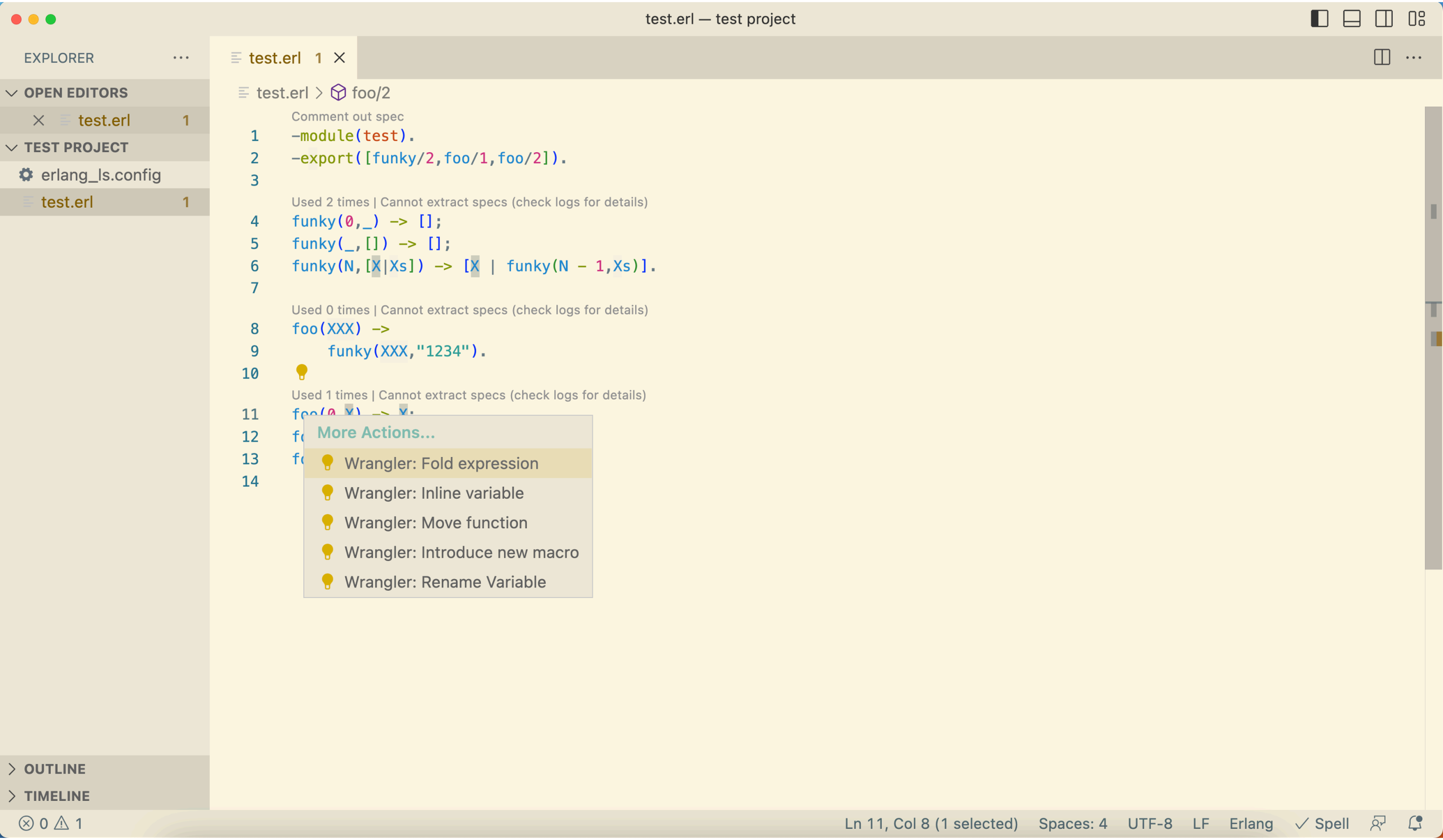
Embed in common IDEs: VS Code, emacs, …

Handle full language, multiple modules, tests, ...

Faithful to layout and comments.

Build in Erlang and apply the tool to itself.


test.erl — test project
EXPLORER
test.erl 1
test.erl > funky/2
OPEN EDITORS
test.erl 1
TEST PROJECT
erlang_ls.config
test.erl 1
Comment out spec
-module(test).
-export([funky/2,foo/1,foo/2]).
Used 2 times | Cannot extract specs (check logs for details)
More Actions...
Wrangler: Fold expression
Wrangler: Move function
(N - 1,Xs)].
logs for details)
foo(XXX) ->
funky(XXX,"1234").
Used 1 times | Cannot extract specs (check logs for details)
foo(0,X) -> X;
foo(_,[]) -> [];
foo(N,[X|Xs]) -> foo(N - 1,Xs).
OUTLINE
TIMELINE
0 1
Ln 4, Col 2
Spaces: 4
UTF-8
LF
Erlang
Spell


test.erl — test project
EXPLORER
test.erl 1
OPEN EDITORS
test.erl 1
TEST PROJECT
erlang_ls.config
test.erl 1
test.erl > foo/2
Comment out spec
1 -module(test).
2 -export([funky/2,foo/1,foo/2]).
3
Used 2 times | Cannot extract specs (check logs for details)
4 funky(0,_) -> [];
5 funky(_,[]) -> [];
6 funky(N,[X|Xs]) -> [X | funky(N - 1,Xs)].
7
Used 0 times | Cannot extract specs (check logs for details)
8 foo(XXX) ->
9 funky(XXX,"1234").
10
Used 1 times | Cannot extract specs (check logs for details)
11 foo(0,X) -> X;
12 fo
13 fo
14
More Actions...
Wrangler: Fold expression
Wrangler: Inline variable
Wrangler: Move function
Wrangler: Introduce new macro
Wrangler: Rename Variable
OUTLINE
TIMELINE
0 1
Ln 11, Col 8 (1 selected)
Spaces: 4
UTF-8
LF
Erlang
Spell

docs

Wrangler Language Server

The Wrangler Language Server is an extension to the Erlang Language Server. It provides Erlang refactorings while implementing the Language Server Protocol through Erlang LS.

You can read more about Wrangler in the Wrangler documentation.

Installation

1. Follow the installation instructions of Wrangler.
2. Install the Erlang Language Server in your chosen text editor.
3. Modify the erlang_ls.config file. Example:

```
...
wrangler:
  enabled: true
  path: "/path/to/wrangler/ebin"
  tab_width: 8
  search_paths: ["folder/where/you/use/refactorings"]
  enabled_refactorings:
    - "comment-out-spec"
    - "fold-expression"
    - "generalise-fun"
    - "move-fun"
    - "new-fun"
    - "new-macro"
    - "new-var"
    - "rename-fun"
    - "rename-var"
    - "inline-var"
```

- Only `enabled` and `enabled_refactorings` are required properties.
- `path` can be omitted if Wrangler is added to the PATH environment variable.
- The folders in `search_paths` will be checked once a refactoring needs to change multiple files in a project. The default value is the project directory. It is recommended to narrow down the search space for large projects.

https://refactoringtools.github.io/docs/wrangler/wls/wrangler_language_server

Haskell	Erlang	OCaml
Strongly typed Lazy Pure + Monads Complex type system Layout sensitive	Weakly typed Strict Some side-effects Concurrency Macros and idioms	Strongly typed Strict Refs etc and i/o. Modules + interfaces Scoping/modules
HaRe	Wrangler	Rotor
Haskell 98 GHC Haskell API … … Alan Zimmerman Basic refactorings, clones, type-based, … Strategic prog	Full Erlang Erlang, syntax_tools HaRe + module, API + DSL, Naive strategic prog	(O)Caml OCaml compiler So far: renaming + dependency theory. Derived visitors

Search-Based Refactoring: Metrics Are Not Enough

Chris Simons[1](✉), Jeremy Singer[2], and David R. White[2]

[1] Depa… echnologies, U… , UK

[2] School of Co… sgow G12 8RZ, UK …c.uk

Automation is highly unlikely to replace the "human in the loop"

Abstract. Search-based Software Engineering (SBSE) techniques have been applied extensively to refactor software, often based on metrics that describe the object-oriented structure of an application. Recent work shows that in some cases applying popular SBSE tools to open-source software does not necessarily lead to an improved version of the software as assessed by some subjective criteria. Through a survey of professionals,

SSBSE 2015

It's just renaming …

what's all the fuss?

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.

A refactoring

```
f x   = … e … e …                f a

f x y = … y … y …                f a e
```

Types sneak in

```
f x = (x*x + 42) + (x + 42)

f x y = (x*x + y) + (x + y)
```

Types sneak in

```
f x = (x*x + 42) + (x + 42)

f x y = (x*x + y) + (x + y)
```

```
funny = length ([[True]] ++ []) +
        length ([True] ++ [])

funny xs = length ([[True]] ++ xs) +
           length ([True] ++ xs)
```

A refactoring

```
f x y = … x … y …
g x y = f x y - 37
```

A refactoring

```
f x y = … x … y …

g x y = f x y - 37

g x y = (… x … y …) - 37
```

A refactoring

```
f x y = if x then y else 42

g x y = f x y - 37

g x y = (if x then y else 42) - 37
```

A refactoring

```
f x = … e … e …
```

A refactoring

```
f x = … e … e …
```

```
f x = … v … v …
      where v = e
```

A refactoring

```
f x = [1 .. x] ++ [1 .. x]
```

```
f x = … v … v …
      where v = [1 .. x]
```

Different roles for atoms

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

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

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

And some peculiarities

```
f1(P) ->
    receive
        {ok, X} -> P!thanks;
        {error,_} -> P!grr
    end,
    P!{value,X}.
```

And some peculiarities

```
f1(P) ->
    receive
        {ok, X} -> P!thanks;
        {error,_} -> P!grr
    end,
    P!{value,X}.
```

```
f2(P) ->
    receive
        {ok, X} -> P!thanks;
        {error,X} -> P!grr
    end,
    P!{value,X}.
```

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))) ;;
```

PLDI 2019

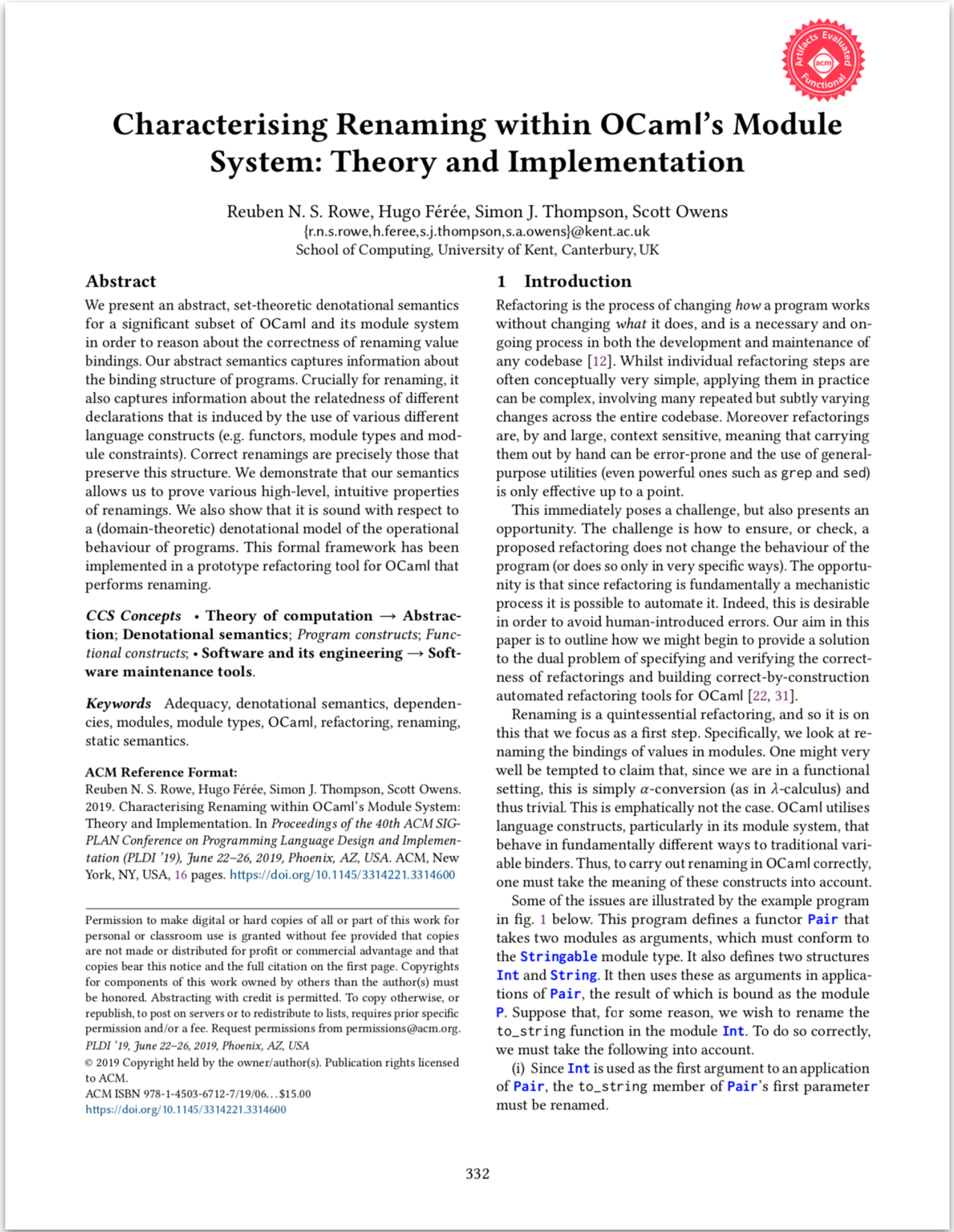
Theory of naming dependency: value extensions.

Characterise renamings by value extension kernels.

Abstract renaming semantics, proved adequate:

"Two equal abstractions have equal concrete versions"

Formalised using Coq.



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

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

Abstract

We present an abstract, set-theoretic denotational semantics for a significant subset of OCaml and its module system in order to reason about the correctness of renaming value bindings. Our abstract semantics captures information about the binding structure of programs. Crucially for renaming, it also captures information about the relatedness of different declarations that is induced by the use of various different language constructs (e.g. functors, module types and module constraints). Correct renamings are precisely those that preserve this structure. We demonstrate that our semantics allows us to prove various high-level, intuitive properties of renamings. We also show that it is sound with respect to a (domain-theoretic) denotational model of the operational behaviour of programs. 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, denotational semantics, dependencies, modules, module types, OCaml, refactoring, renaming, static semantics.

ACM Reference Format:
Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, Scott Owens. 2019. Characterising Renaming within OCaml's Module System: Theory and Implementation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314600

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314600

1 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 [12]. 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, this 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-calculus) 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 functor `Pair` that takes 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`. Suppose that, for some reason, we wish to rename the `to_string` function in the module `Int`. To do so correctly, we must take the following into account.

(i) Since `Int` is used as the first argument to an application of `Pair`, the `to_string` member of `Pair`'s first parameter must be renamed.

Why have you messed up the layout of my program?

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
    ]
```

Appearance must be right

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

```
{v1, v2, v3}
```

```
{v1,v2,v3}
```

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

```
f (g x y)
```

```
f $ g x y
```

Appearance must be right

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

```
{v1, v2, v3}
```

```
{v1,v2,v3}
```

```
data MyType = Foo |
              Bar |
              Baz
```

```
data HerType = Foo
             | Bar
             | Baz
```

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

```
f (g x y)
```

```
f $ g x y
```

Preserving appearance

Preserve precisely parts not touched.

Pretty print … or use lexical details.

giffgaff 4G 17:36 65%

Home

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...

4 7 40

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.

3 4 26

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...

4 7 40

Don Stewart @donsbot · 3h

We have data showing how much faster code review is when format is removed

Why should I trust your refactoring tool on my project?

POINT

Refactoring Tools Are Trustworthy Enough

John Brant

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 great feature to have, it's neither a necessary nor sufficient 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 revert 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

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

COUNTERPOINT

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 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.

There's also a third perspective—the one I care about most. As an engineer, and even more so as a researcher, I want to do things that are state-of-the-art. Where the state-of-the-art leaves something to be desired, I want to push it further. If that's impossible, I want to know why, and I want people to understand why so that they can adjust their expectations. Refactoring-tool users will more easily accept limitations if these limitations are inherent in the nature of the matter and aren't engineering shortcomings.

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 expla-

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

IEEE Software, Nov/Dec 2015

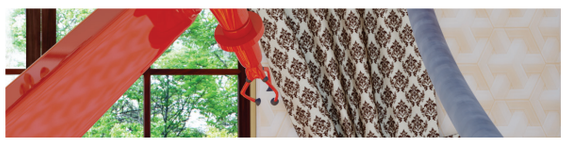
Challenges to and Solutions for Refactoring Adoption

An Industrial Perspective

Tushar Sharma and Girish Suryanarayana, 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.[1] 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."[2] Martin Fowler's seminal work increased refactoring's popularity and extended its academic and industrial reach.[3] Modern software development methods such as Extreme Programming ("refactor mercilessly")[4] have adopted refactoring as an essential element.

However, our experience assessing industrial software design[5] 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.

0740-7459/15/$31.00 © 2015 IEEE

IEEE Software, Nov/Dec 2015

Breaking code

Cannot justify the time spent

Unpredictable impact

Difficult to review

Inadequate tools

Why should I trust your refactoring tool on my project?

Answer 1:

Evidence that it is built right

Building trust

Open Source … confidence in the code … other committers.

Openness of the system …

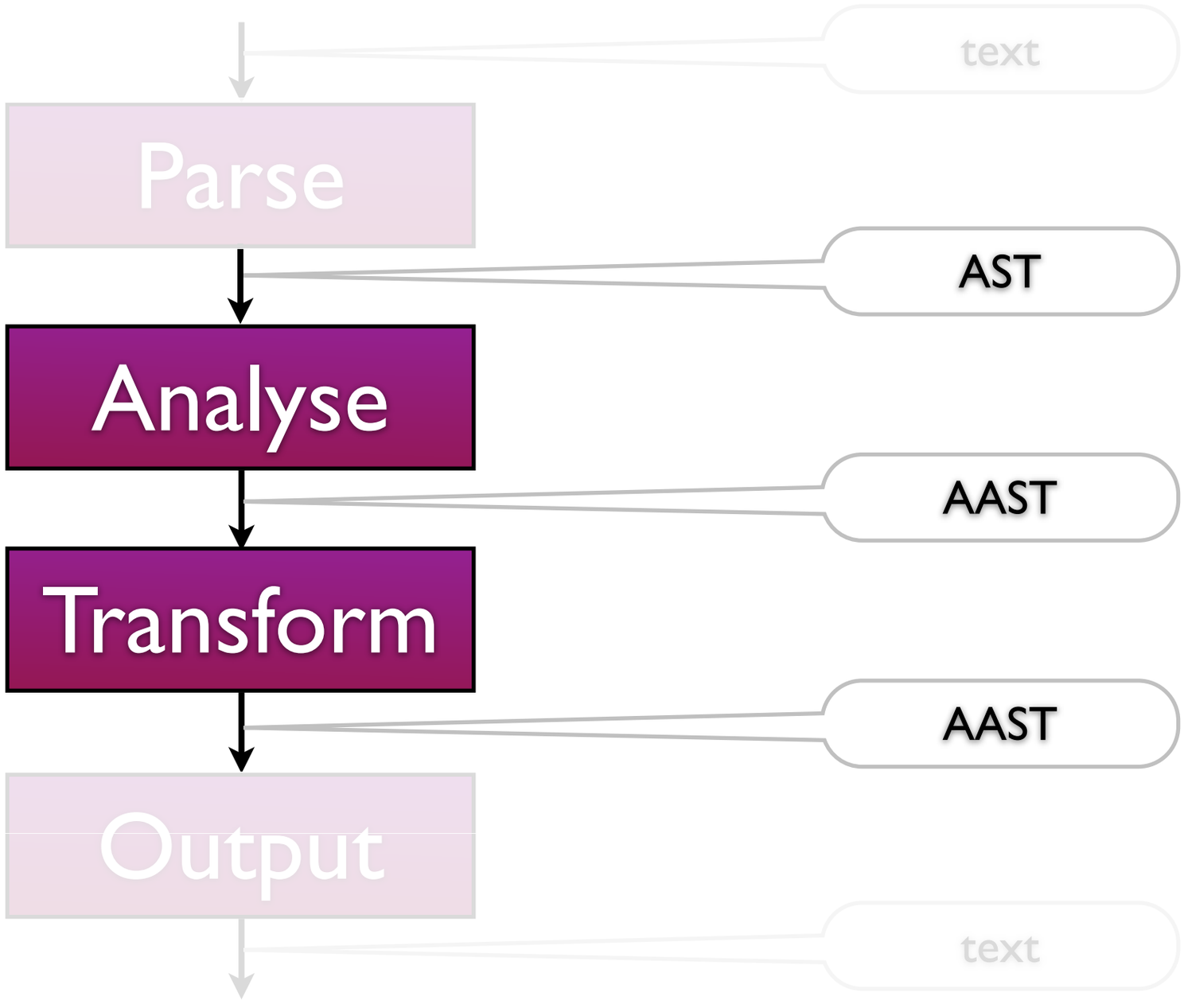
… you can check the changes that a refactoring makes,

… and for the DSL can see which refactorings performed.

GHC *vs* Haskell standards *vs* other Haskell implementations.

Editor and IDE integration

```
-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).
```

Traversals, strategies and visitors

Multi-purpose

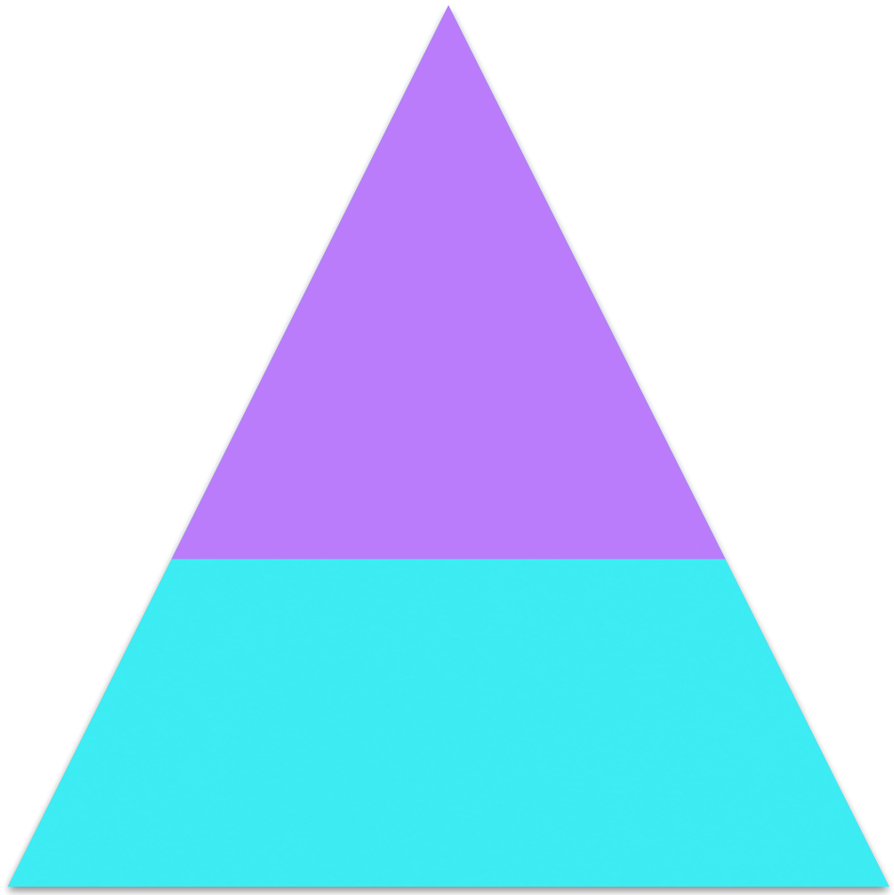
Collect and analyse info.

Effect a transformation.

Separation of concerns

Point-wise operation …

… and tree traversal



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

Insight: there are patterns more general than individual refactorings

Layers, libraries and languages?

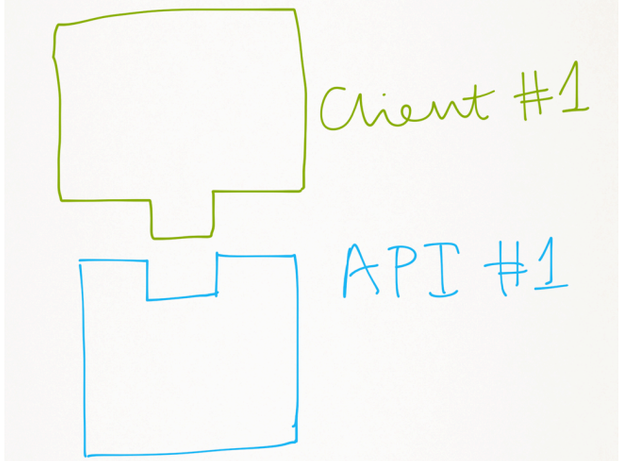
Clone detection and removal

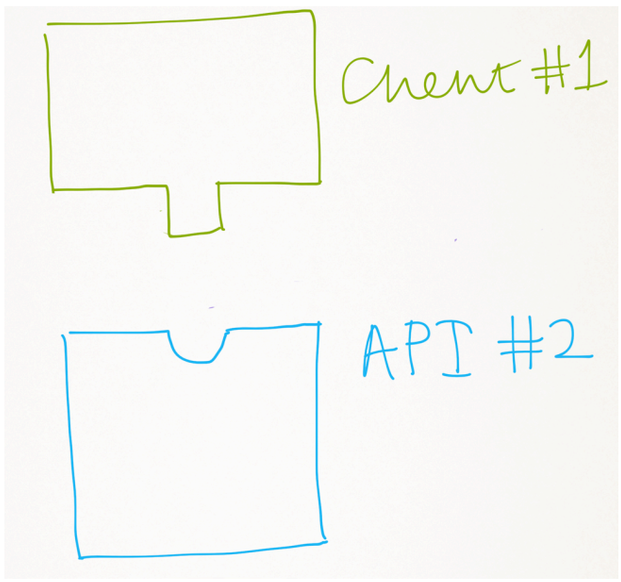
Module structure improvement

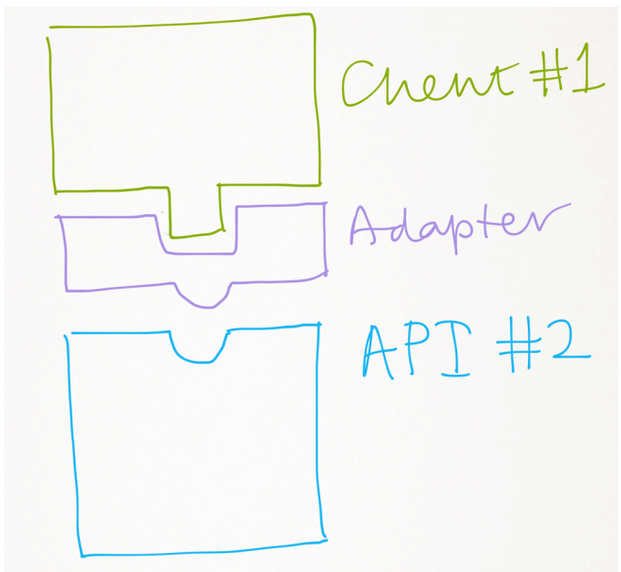
DSL for composite refactorings

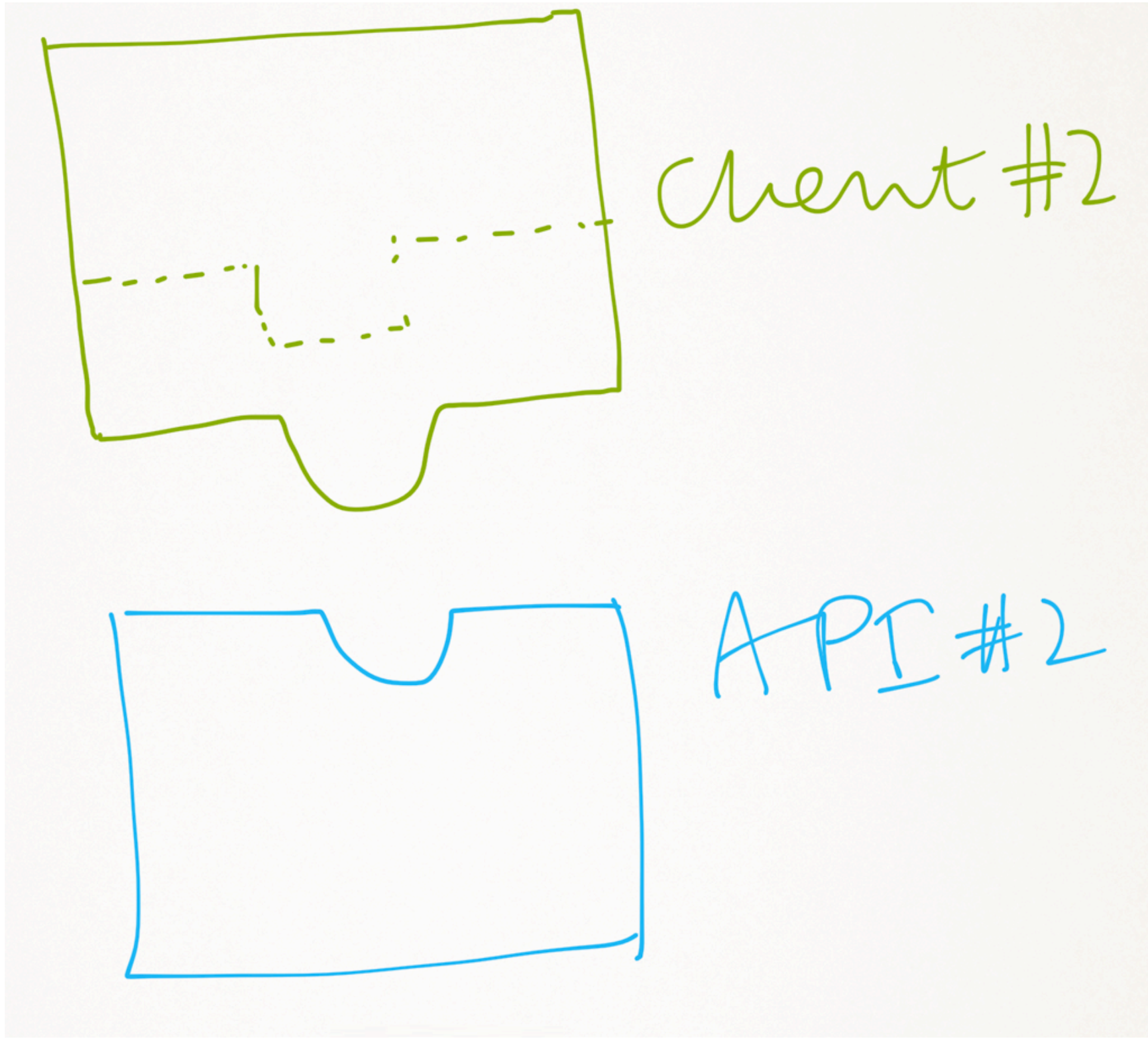
API: define new refactorings

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


Client #1
API #1


Client #1
API #2


Client #1
Adapter
API #2


Client #2
API #2

Function pattern

- definition

- use

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

Function pattern
- definition

- use

```
f x = x+1

g y = f (y+2) - f (y-2)
```

Renaming

```
h x = x+1

g y = h (y+2) - h (y-2)
```

Function pattern
- definition
- use

```
f x = x+3

g z = map f z
```

Generalisation

```
f x y = x+y

g z = map (\x -> f x 3) z
```

Implementation

- *define the old function using the new*
- *substitute this definition*

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
h x = x+1

f = \x -> h x
```

```
g y = f(y+2) - f(y-2)
```

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

f = \x -> h x
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
g y = f(y+2) - f(y-2)
```

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
h x = x+1

f = \x -> h x

g y = (\x -> h x)(y+2)
      - f(y-2)
```

Implementation

- *define the old function using the new*
- *substitute this definition*
- *rewrite to simplify*

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
h x = x+1

f = \x -> h x

g y = (\x -> h x)(y+2)
      - f(y-2)
```

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

f = \x -> h x
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
g y = h(y+2) - f(y-2)
```

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
h x = x+1

f = \x -> h x
```

```
g y = h(y+2)   …
```

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

f = \x -> h x
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
g y = h(y+2) - h(y-2)
```

Renaming

```
f x = x+1

g y = f (y+2) - f (y-2)
```

```
h x = x+1

g y = h (y+2) - h (y-2)
```

```
h x = x+1

g y = h(y+2) - h(y-2)
```

Implementation

- *define the old function using the new*
- *substitute this definition*
- *rewrite to simplify*

Refactoring = *Substitution* + *Rewriting*

Refactoring = Substitution + Rewriting

Towards Generic, Language-Independent Refactorings

Simon Thompson[1]
School of Computing, University of Kent, Canterbury, UK
Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

Dániel Horpácsi
Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

Abstract

Eelco Visser's work has always encouraged stepping back from the particular to look at the underlying, conceptual problems.

In that spirit we present an approach to describing refactorings that abstracts away from particular refactorings to classes of similar transformations, and presents an implementation of these that works by substitution and subsequent rewriting.

Substitution is language-independent under this approach, while the rewrites embody language-specific aspects. Intriguingly, it also goes back to work on API migration by Huiqing Li and the first author, and sets refactoring in that general context.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases refactoring, generic, language independent, rewriting, substitution, API upgrade

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.26

Why should I trust your refactoring tool on my project?

Answer 2:

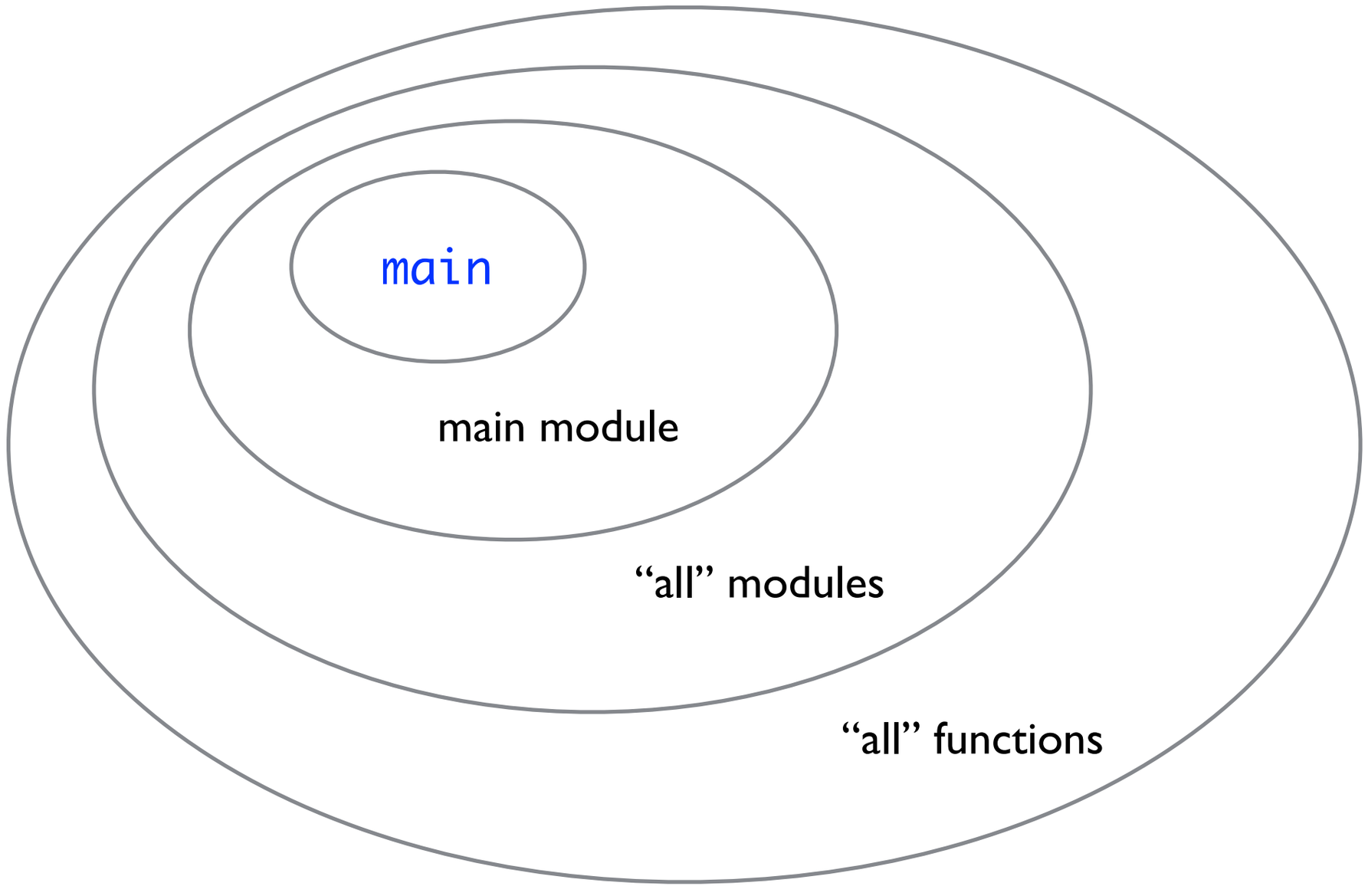
Evidence that it behaves correctly

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



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

	test	verify
instances of the refactoring		
the refactoring itself		

Assuring meaning preservation

	test	verify
instances of the refactoring	Rename `foo` to `bar` in this project.	
the refactoring itself		

Assuring meaning preservation

	test	verify
instances of the refactoring	Rename `foo` to `bar` in this project.	
the refactoring itself	Renaming for all names, functions and projects.	

	test	verify
instances of the refactoring	✓	✓
the refactoring itself	✓	✓

Testing instances and refactorings

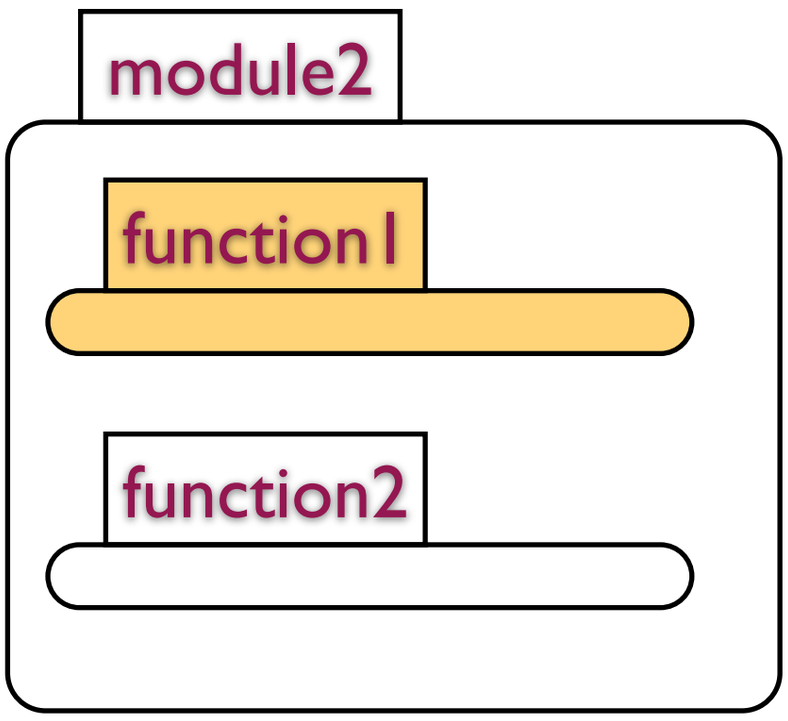
	test	verify
instances of the refactoring	✓	
the refactoring itself		

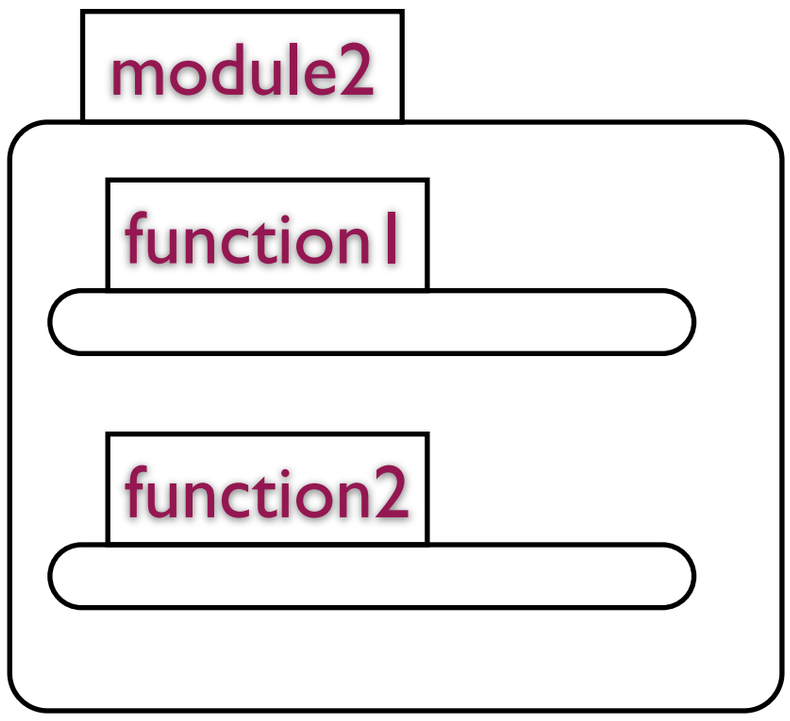
Testing new *vs* old (with Huiqing Li)

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).





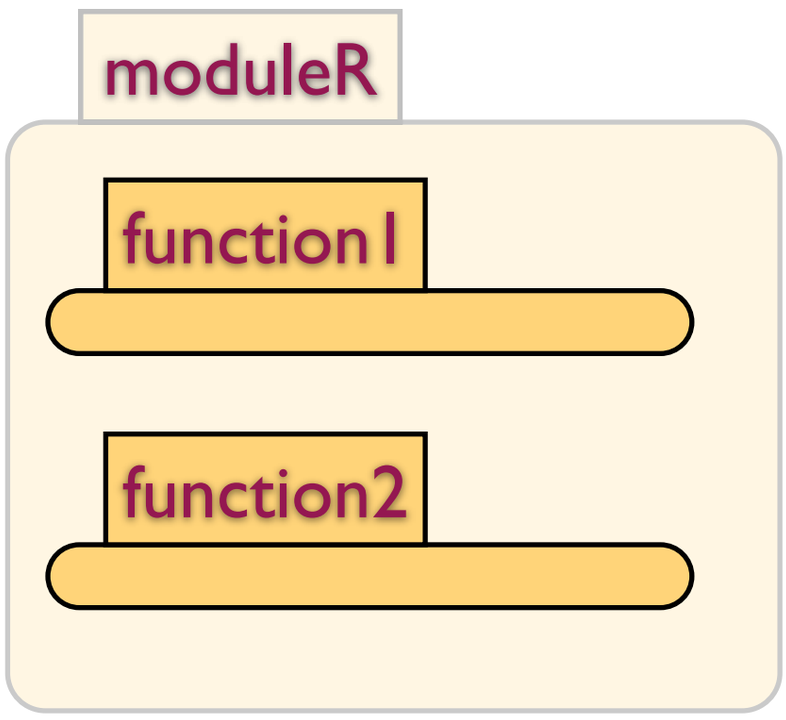
	test	verify
instances of the refactoring		
the refactoring itself	✓	

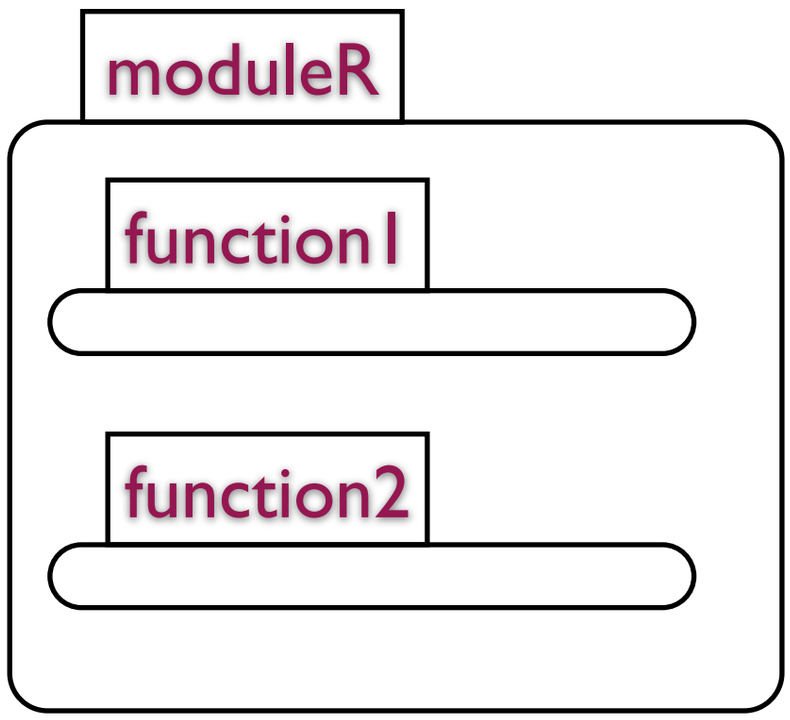
Fully random

Generate random modules,

… generate random refactoring commands,

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





Verify a complete refactoring “by hand”

	test	verify
instances of the refactoring		
the refactoring itself		✓

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.

Refactoring = Substitution + Rewriting

Verification

Does old = new?
 - one for each refactoring

Are the rewrites ok?
 - once for all refactorings

```
f x = x+1

g y = f(y+2) - f(y-2)

h x = x+1

f = \x -> h x
```

Automated proof for a refactoring instance

	test	verify
instances of the refactoring		✓
the refactoring itself		

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

Example

```
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

```
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
```

```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
```

```
{-# 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
```

```
-- 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
```

```
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
```

```
-- 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
```

```
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' :: 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
```

	test	verify
instances of the refactoring	✓	✓
the refactoring itself	✓	✓

Refactoring = *Substitution* + *Rewriting*

Function application in Erlang

```
f(X) -> X+1.

g(Xs) -> map(fun f/1,Xs).

h(Y) -> spawn(mod,f,[Y]).
```

Function application in Erlang and Haskell

```
f(X) -> X+1.

g(Xs) -> map(fun f/1,Xs).

h(Y) -> spawn(mod,f,[Y]).
```

```
f x y = x+y

g z = z `f` (z-1)

h x xs = map (x `f`) xs
```

Generic *Specific*

Refactoring = *Substitution* + *Rewriting*

Why should I trust your refactoring tool?

Build it right

- architecture
- traversals
- general patterns
- APIs
- DSLs

Provide assurance

- heterogeneous
- test and verify
- instances and …
- … full refactorings
- substitution + rewriting

Questions?

https://github.com/alanz/HaRe

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

https://gitlab.com/trustworthy-refactoring/refactorer

More complex refactorings

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.

3 8

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.

2

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.

1 2

Replace lists with “Hughes lists”

```haskell
explode :: [a] -> [a]
explode lst = concat (map (\x -> replicate (length lst) x) lst)
```

Replace lists with “Hughes lists”

```haskell
explode :: [a] -> [a]
explode lst = concat (map (\x -> replicate (length lst) x) lst)
```

```haskell
explode :: DList a -> DList a
explode lst =
    DL.concat
        (DL.map
            (\x -> DL.replicate (length (DL.toList lst)) x) lst)
```

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
```

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 refactorings?

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)
```