Refactoring for Functional Programs

Simon Thompson, University of Kent

What have we learned about tool building?

Simon Thompson, University of Kent


Huiqing Li


Colin Runciman


Thomas Arts

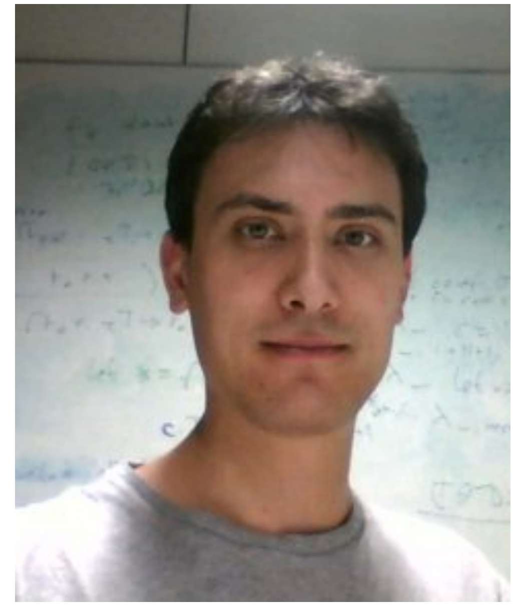

Dániel Horpácsi


Judit Kőszegi


Nik Sultana

Scott Owens

Reuben Rowe



Hugo Férée

Chris Brown

György Orosz

Melinda Tóth

Stephen Adams

Andreas Reuleaux

Claus Reinke

Pablo Lamela

Jane Street

Science

Engineering

Human factors

Science

Engineering

Human factors

Usability & Trust

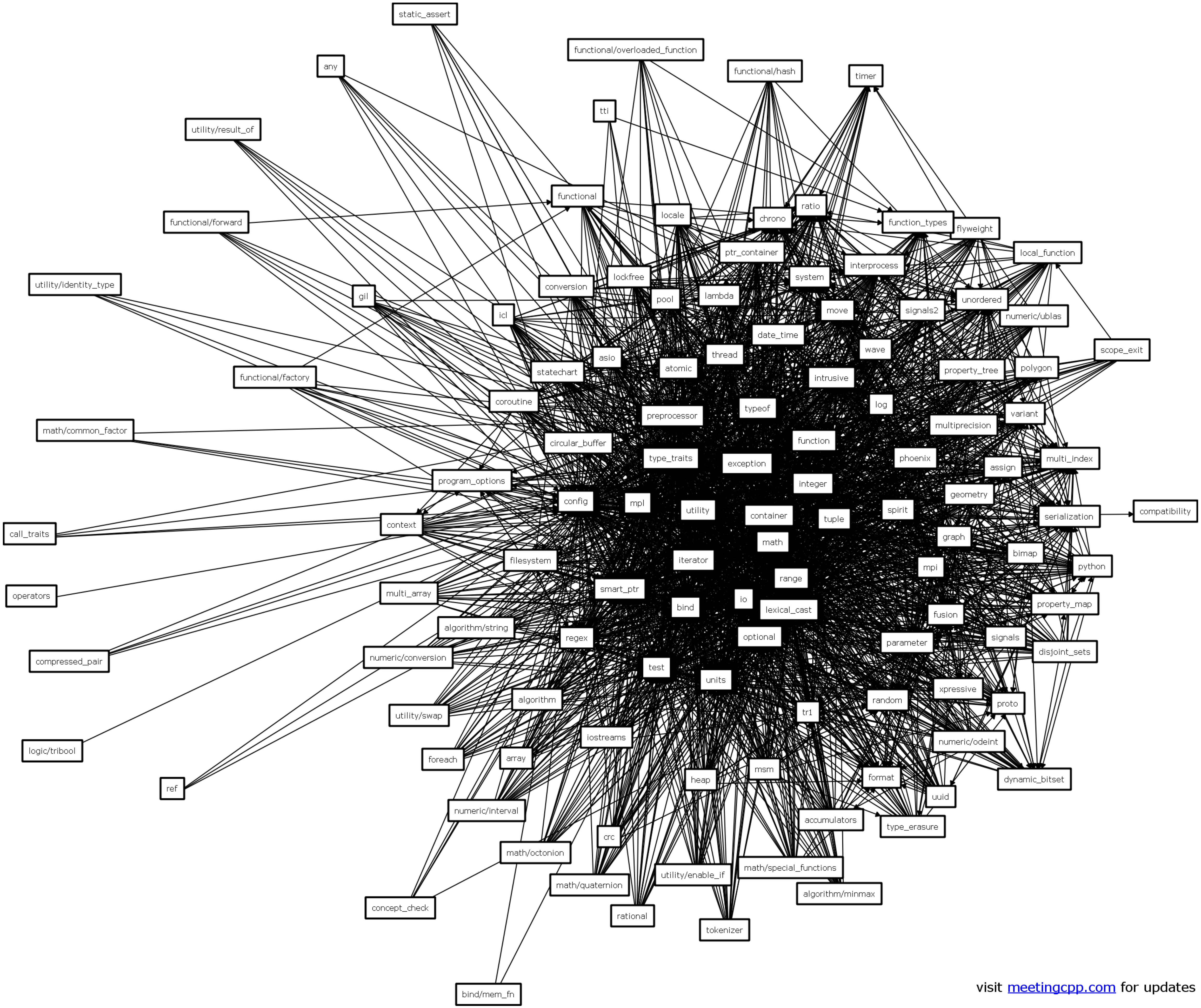
Automation

Languages

What do you mean by “refactoring”?

3 src/EqSolve.hs View

```diff
@@ -187,11 +187,12 @@ splitOrConvert (m, r, c) sol =
     Nothing -> Nothing
 
 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
 
```



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:

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 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

```
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

```
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},
```

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.

Refactoring tools

Refactoring

=

Transformation

Refactoring

=

Transformation

Refactoring

=

Transformation + Pre-condition

How to refactor?

By hand … using an editor

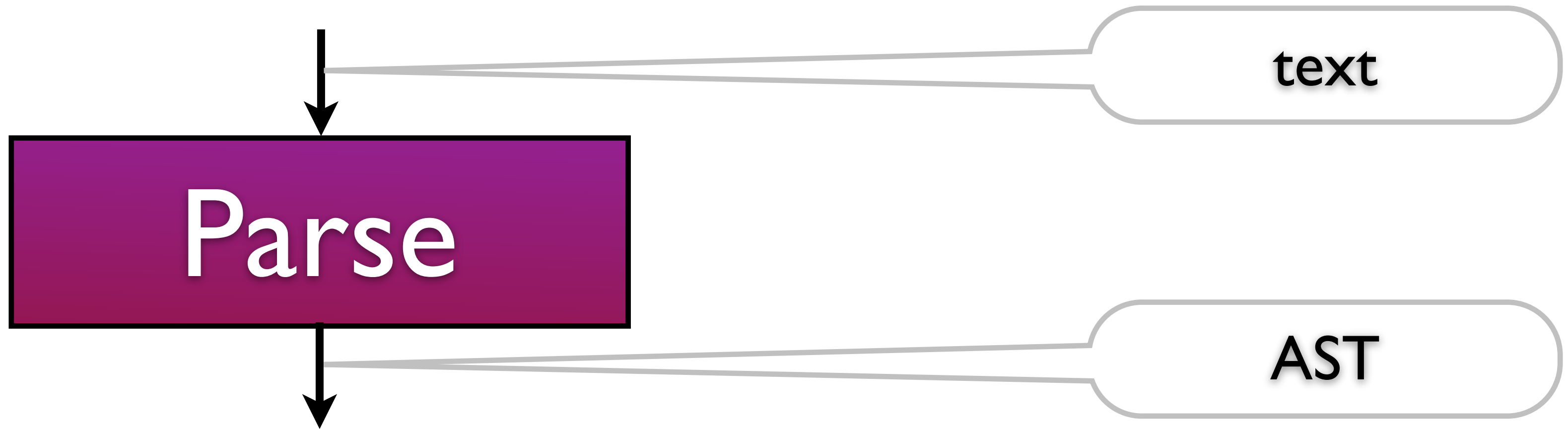
- Flexible … but error-prone.
- Infeasible in the large.

Tool-supported

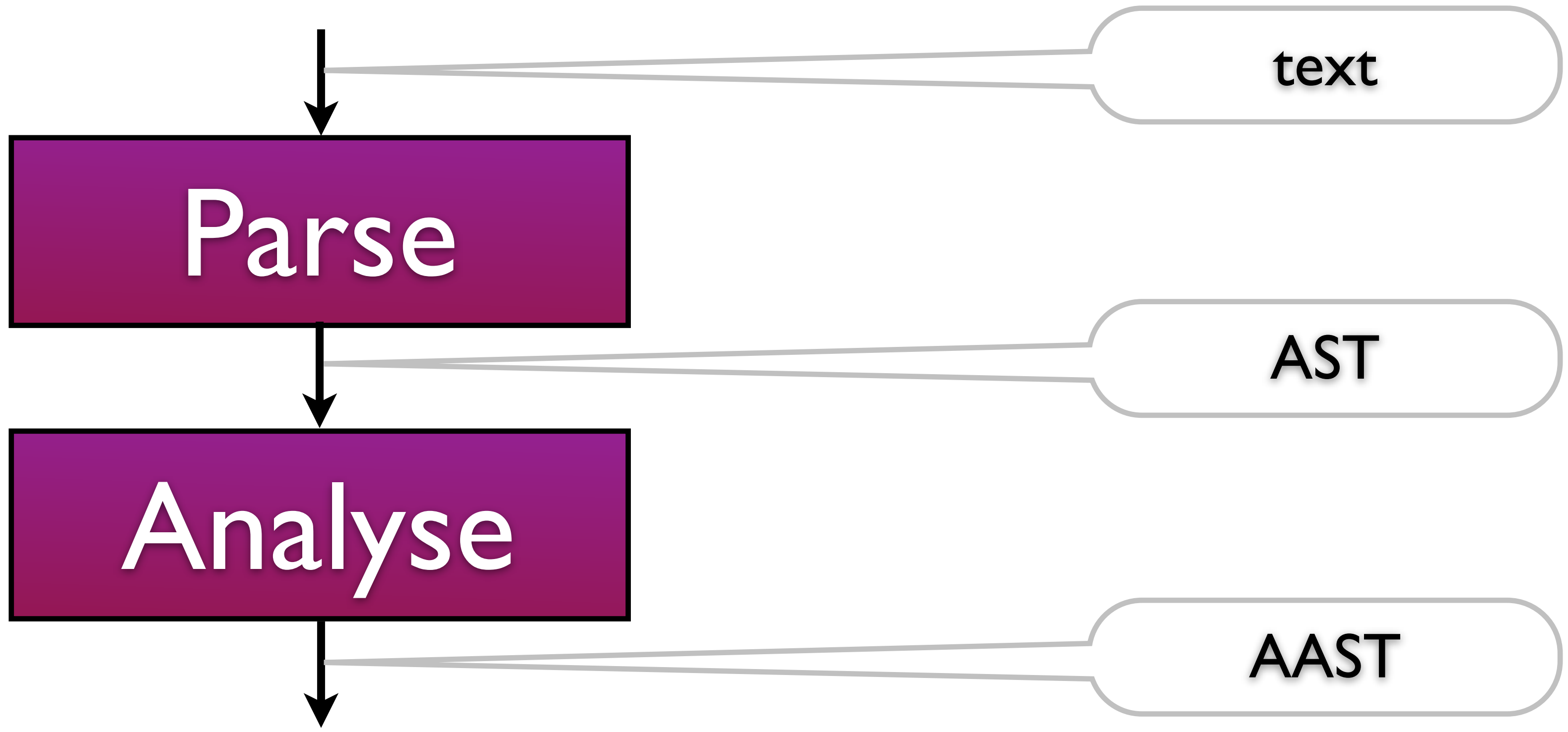
- Handles transformation *and* analysis.
- Scalable to large-code bases: module-aware.
- Integrated with tests, macros, ...

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

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

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

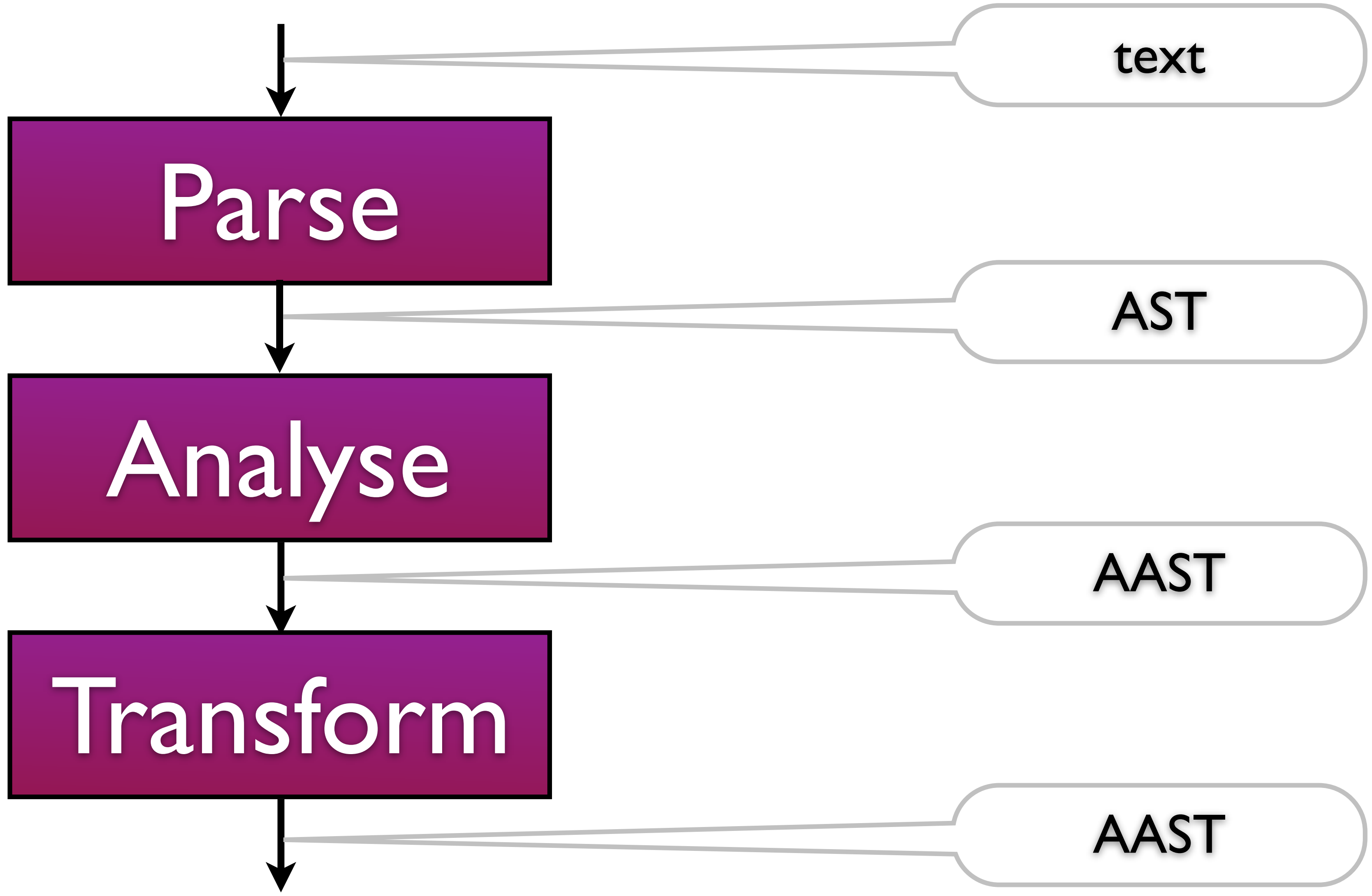
```
-module(foo).
-export([foo/1,foo/0]).

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

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

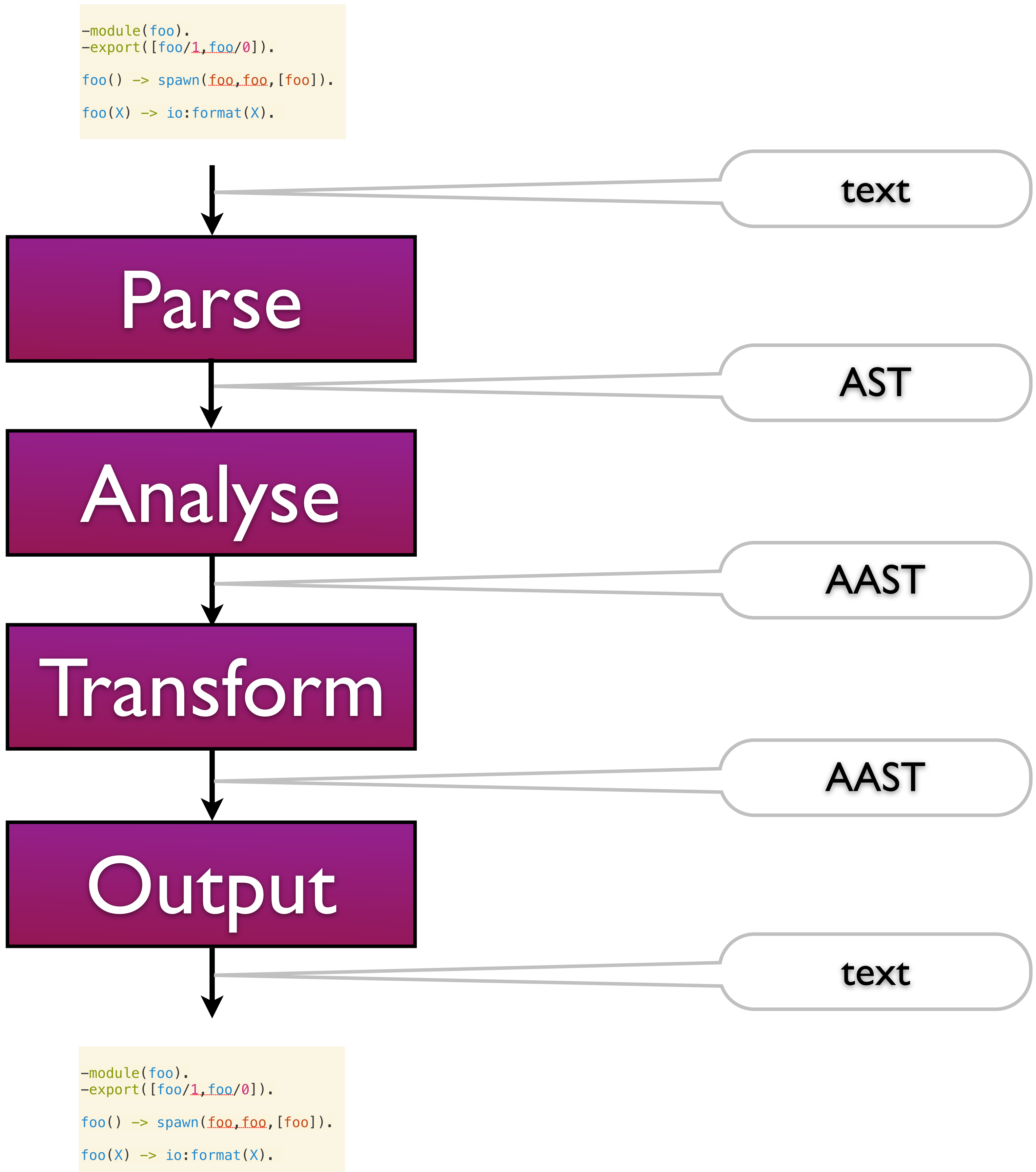


```
-module(foo).
-export([foo/1,foo/0]).

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).
text
Parse
AST
Analyse
AAST
Transform
AAST
Output
text
-module(foo).
-export([foo/1,foo/0]).
foo() -> spawn(foo,foo,[foo]).
foo(X) -> io:format(X).

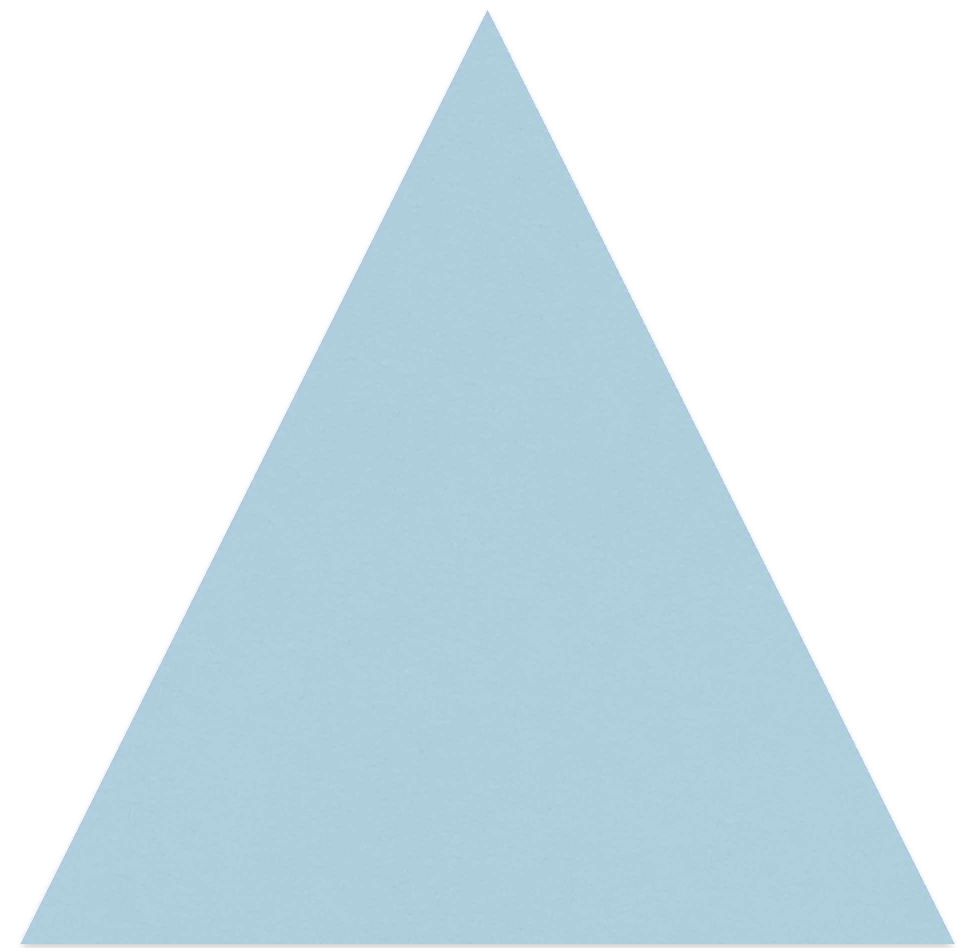
Traversals, strategies and visitors

Multi-purpose

- Collect and analyse info.
- Effect a transformation.

Separation of concerns

- Point-wise operation …
- … and tree traversal



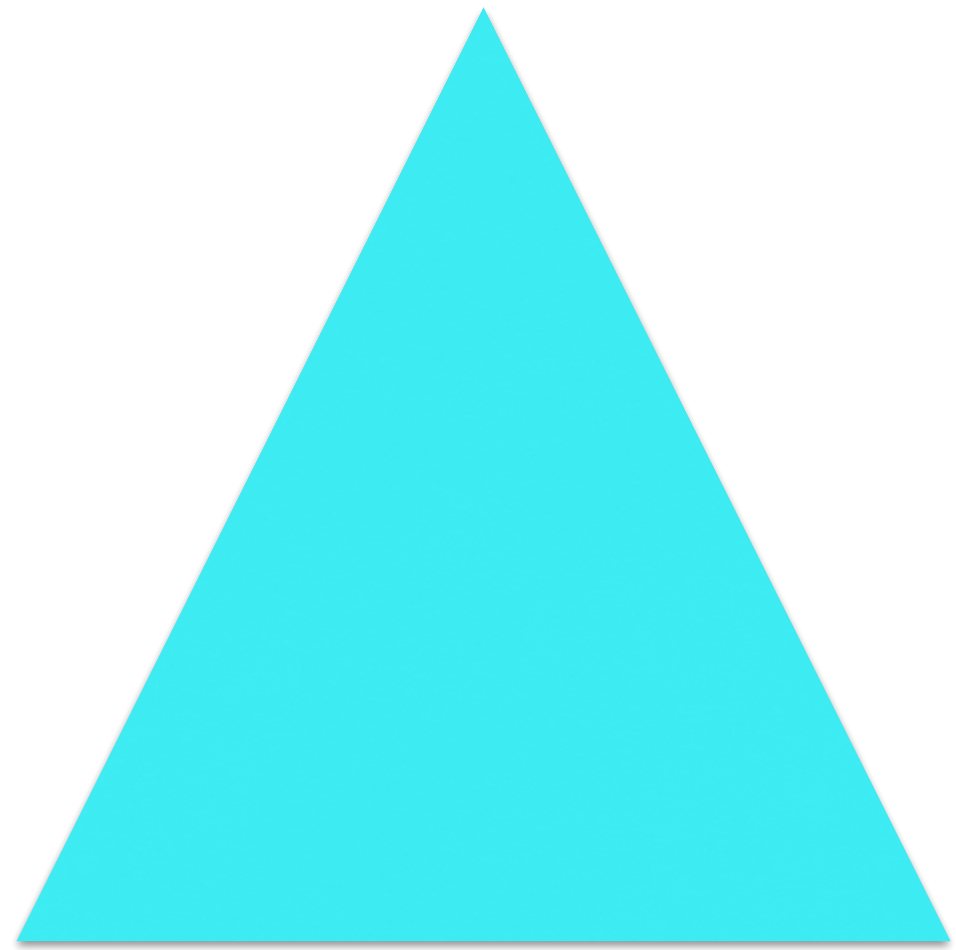
Traversals, strategies and visitors

Multi-purpose

- Collect and analyse info.
- Effect a transformation.

Separation of concerns

- Point-wise operation …
- … and tree traversal



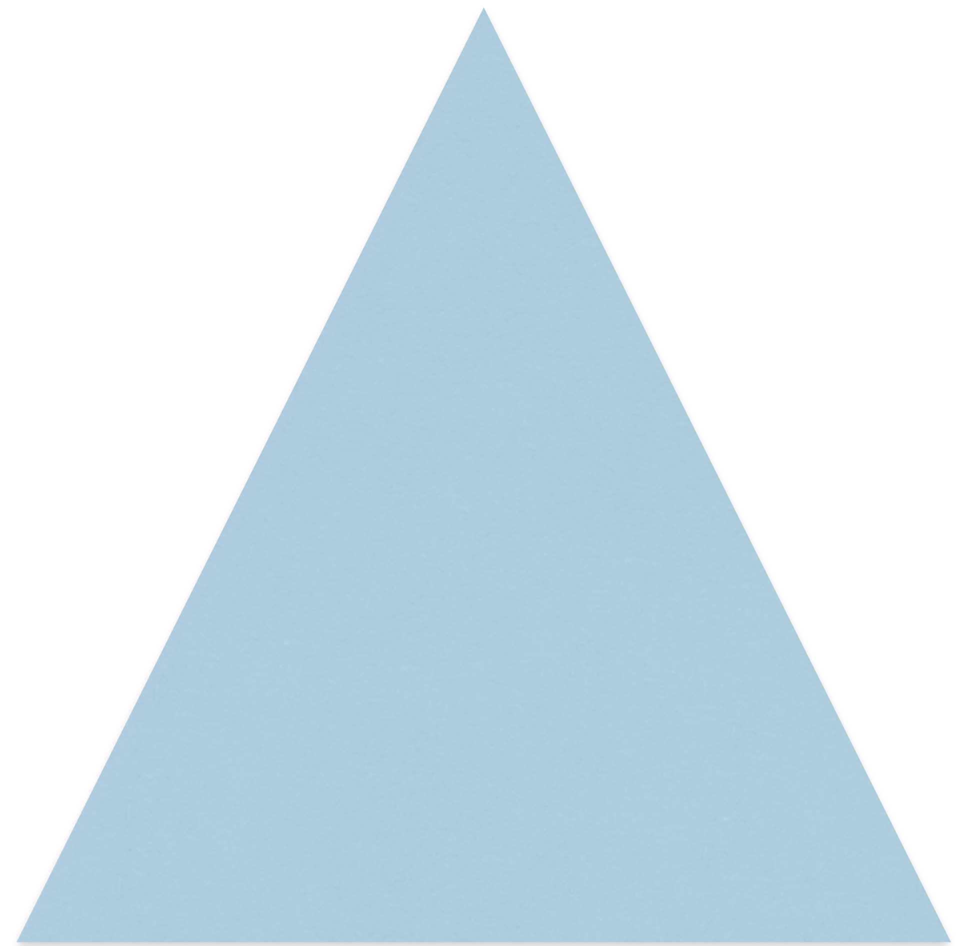
Traversals, strategies and visitors

Multi-purpose

- Collect and analyse info.
- Effect a transformation.

Separation of concerns

- Point-wise operation …
- … and tree traversal



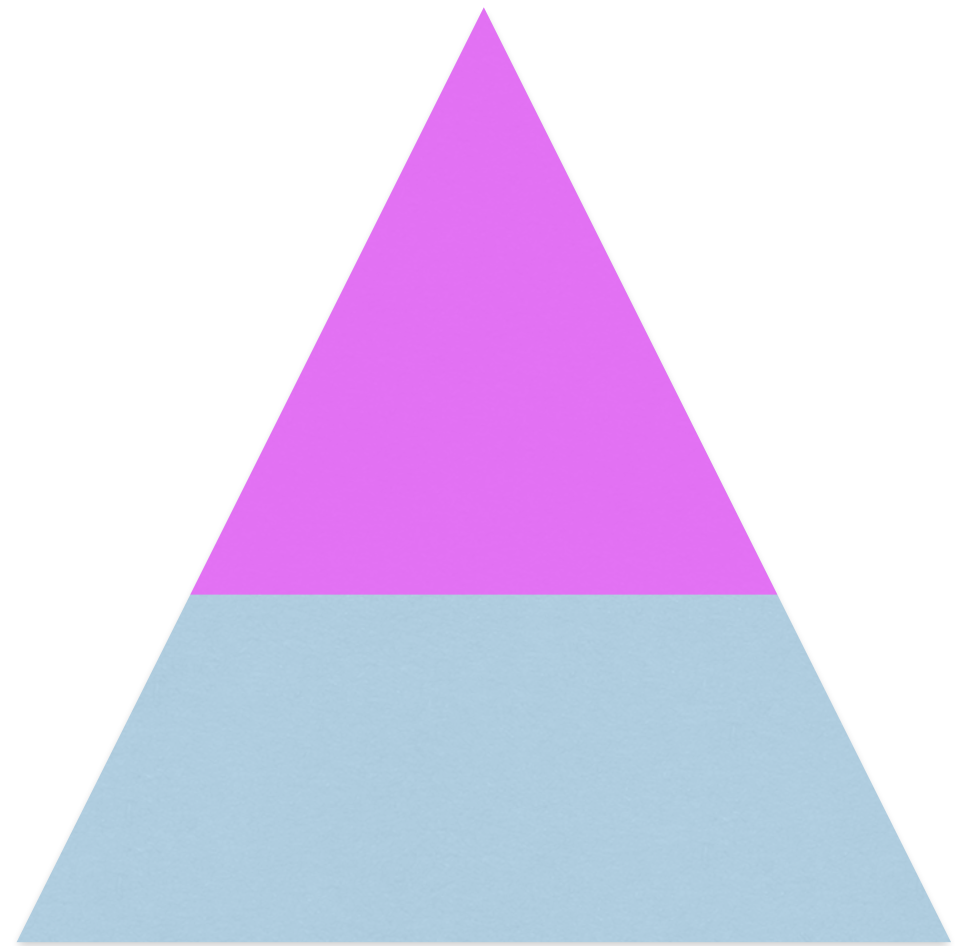
Traversals, strategies and visitors

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

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang

Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

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

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

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
Haskell 98 Programmatica / GHC Haskell API Basic refactorings, clones, type-based, … Strategic prog	Full Erlang Erlang, syntax_tools HaRe + module, API, DSL, context. Naive strategic prog

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 Programmatica / GHC Haskell API Basic refactorings, clones, type-based, … Strategic prog	Full Erlang Erlang, syntax_tools HaRe + module, API, DSL, context. Naive strategic prog	(O)Caml OCaml compiler So far: renaming & dependency theory. Derived visitors

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

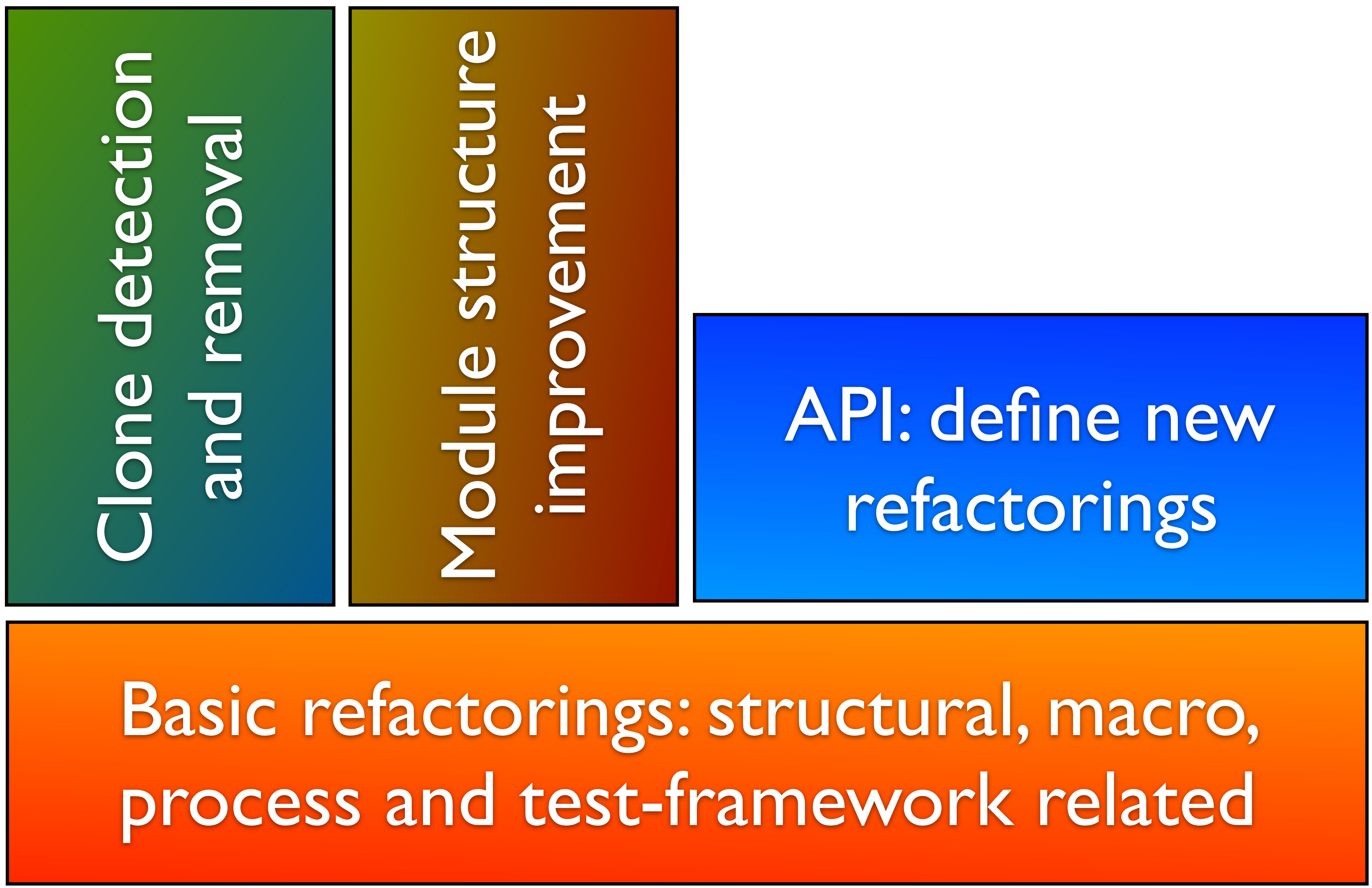
Wrangler

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

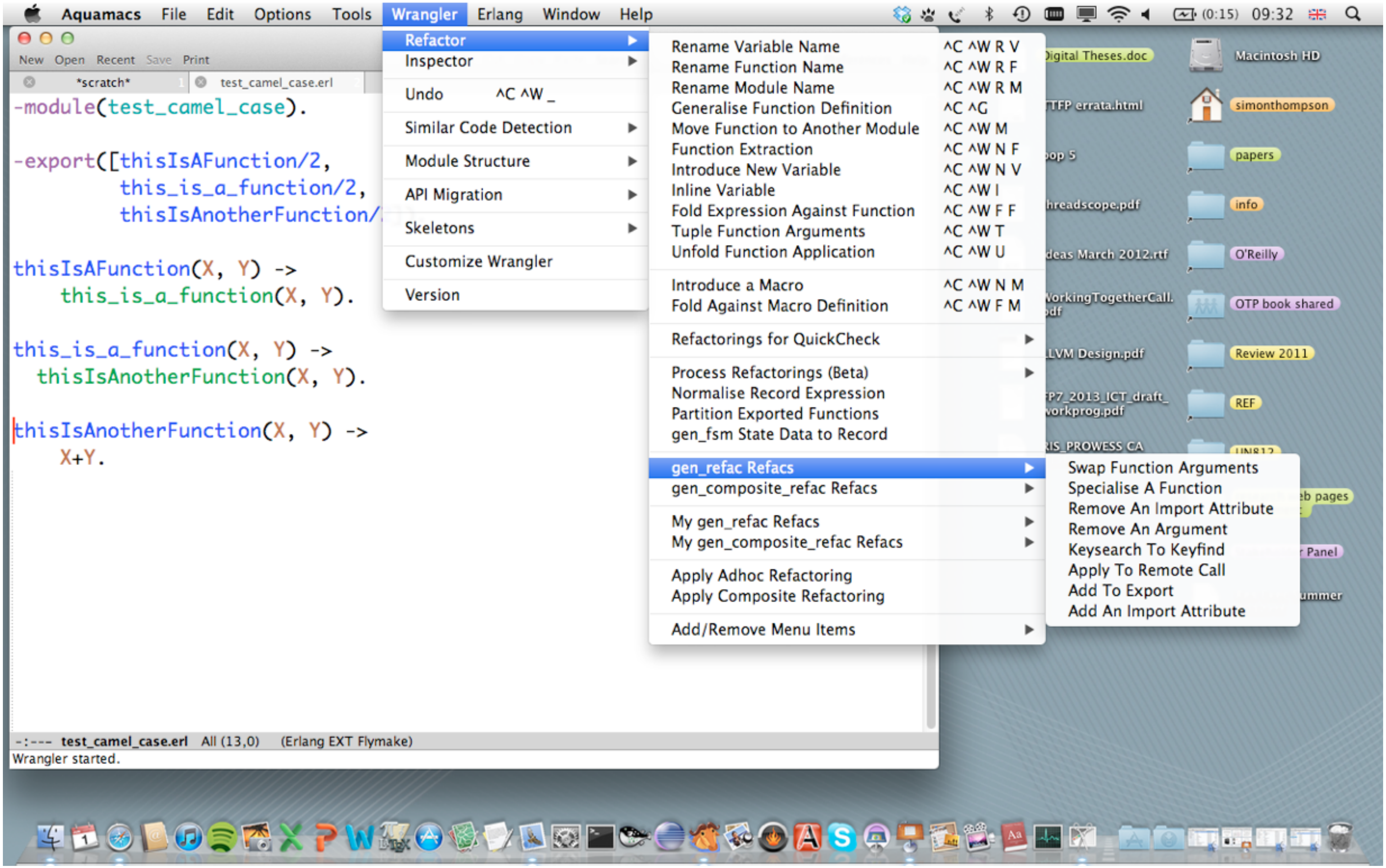
Clone detection and removal

Module structure improvement

DSL for composite refactorings

API: define new refactorings

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


Aquamacs File Edit Options Tools Wrangler Erlang Window Help
(0:15) 09:32
New Open Recent Save Print
scratch
test_camel_case.erl
-module(test_camel_case).
-export([thisIsAFunction/2,
this_is_a_function/2,
thisIsAnotherFunction/
thisIsAFunction(X, Y) ->
this_is_a_function(X, Y).
this_is_a_function(X, Y) ->
thisIsAnotherFunction(X, Y).
thisIsAnotherFunction(X, Y) ->
X+Y.
-:--- test_camel_case.erl All (13,0) (Erlang EXT Flymake)
Wrangler started.
Refactor
Inspector
Undo ^C ^W _
Similar Code Detection
Module Structure
API Migration
Skeletons
Customize Wrangler
Version
Rename Variable Name ^C ^W R V
Rename Function Name ^C ^W R F
Rename Module Name ^C ^W R M
Generalise Function Definition ^C ^G
Move Function to Another Module ^C ^W M
Function Extraction ^C ^W N F
Introduce New Variable ^C ^W N V
Inline Variable ^C ^W I
Fold Expression Against Function ^C ^W F F
Tuple Function Arguments ^C ^W T
Unfold Function Application ^C ^W U
Introduce a Macro ^C ^W N M
Fold Against Macro Definition ^C ^W F M
Refactorings for QuickCheck
Process Refactorings (Beta)
Normalise Record Expression
Partition Exported Functions
gen_fsm State Data to Record
gen_refac Refacs
gen_composite_refac Refacs
My gen_refac Refacs
My gen_composite_refac Refacs
Apply Adhoc Refactoring
Apply Composite Refactoring
Add/Remove Menu Items
Swap Function Arguments
Specialise A Function
Remove An Import Attribute
Remove An Argument
Keysearch To Keyfind
Apply To Remote Call
Add To Export
Add An Import Attribute
Macintosh HD
simonthompson
papers
info
O'Reilly
OTP book shared
Review 2011
REF

Analyses needed …

Static semantics

Types

Modules

Side-effects

Analyses needed …

Static semantics

Types

Modules

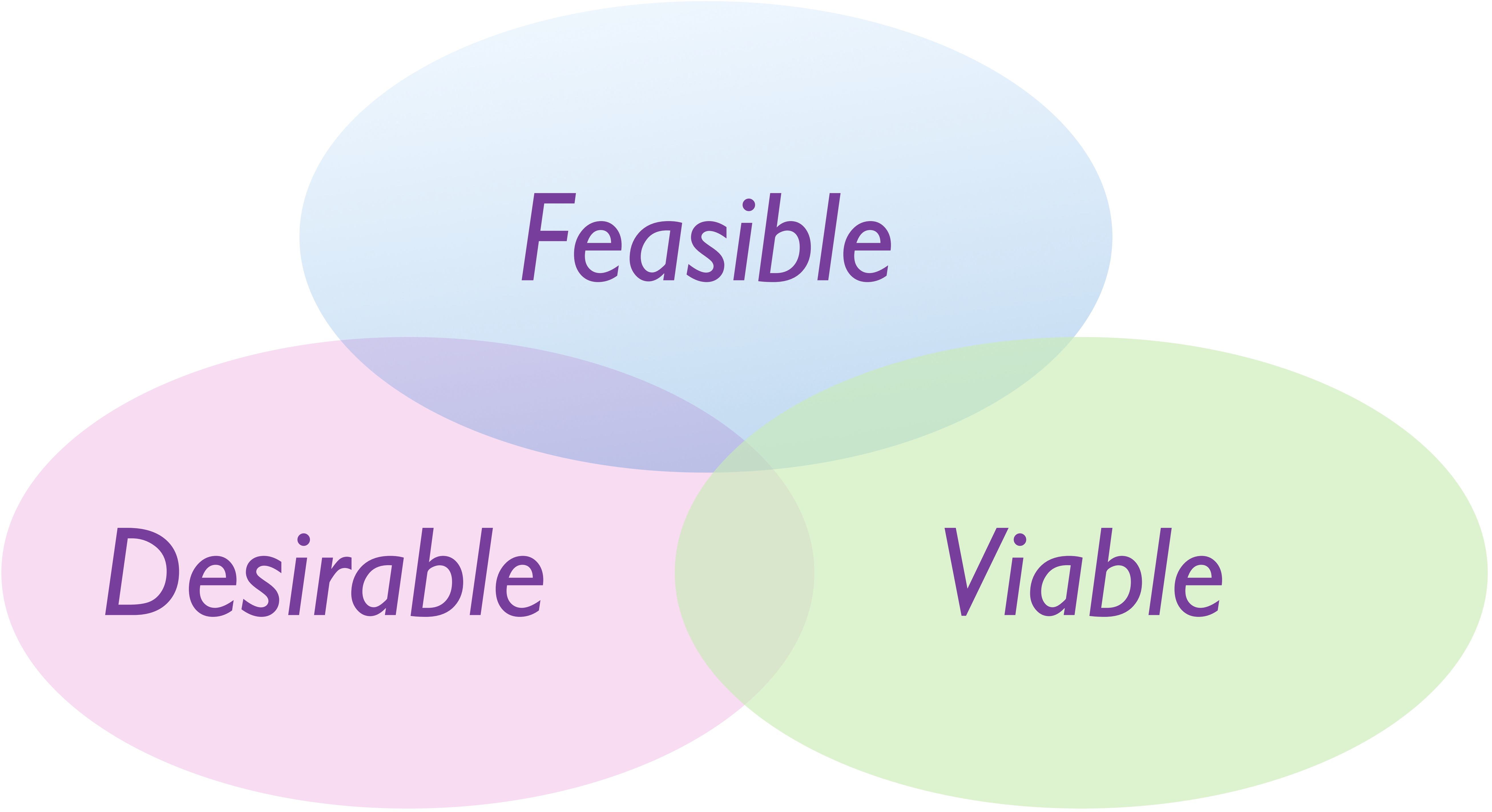
Side-effects

Atoms

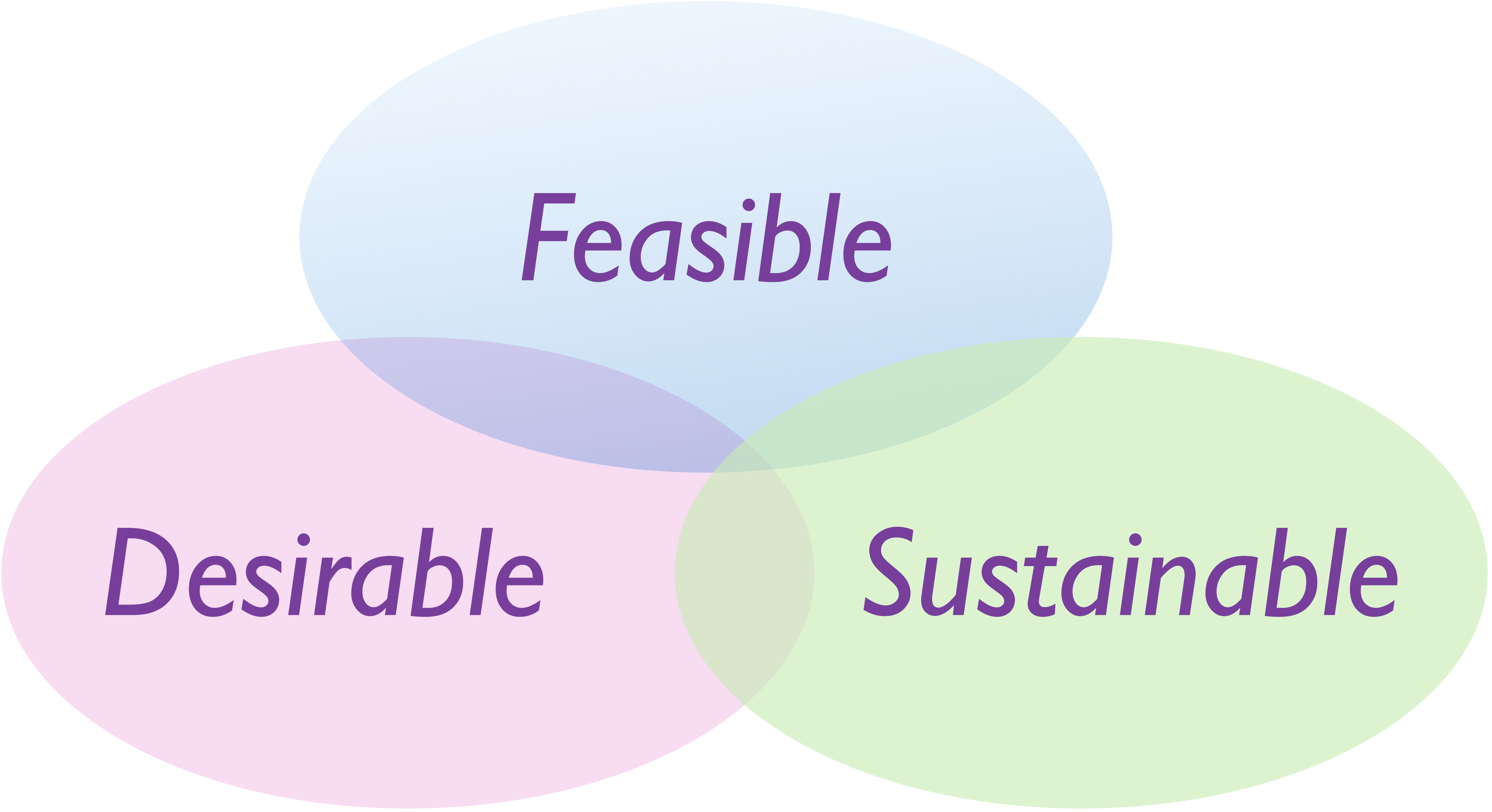
Process structure

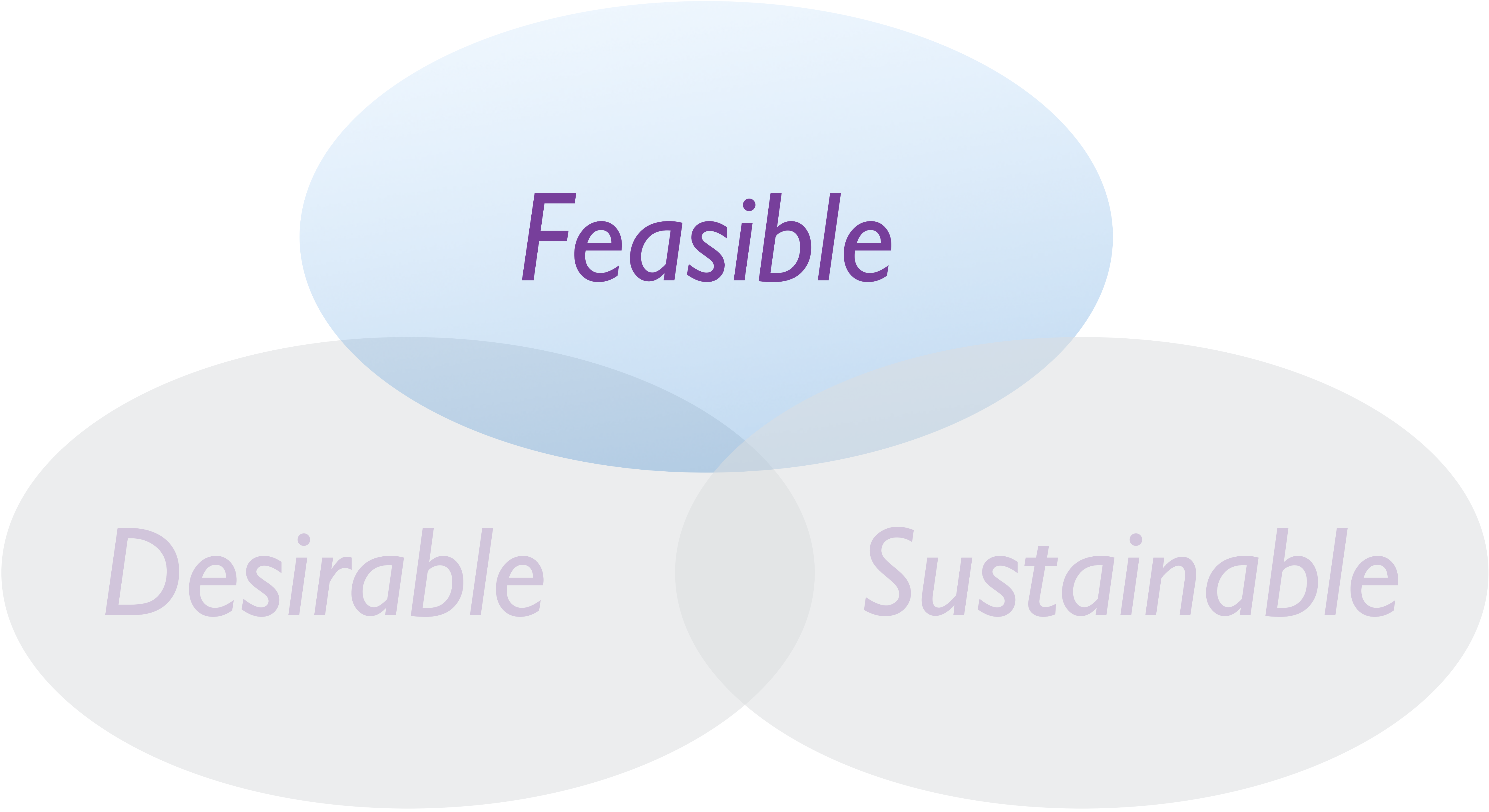
Macros

Conventions and frameworks



dschool.stanford.edu


Feasible
Desirable
Sustainable



Feasible
Desirable
Sustainable

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*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)
```

✗

… as do different sorts of 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}.
```

✓

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

```
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 (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

```
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))) ;;
```

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
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 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. functors, module types and module constraints). Correct renamings 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 renamings. 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:
Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, and 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

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 that even 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` 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,

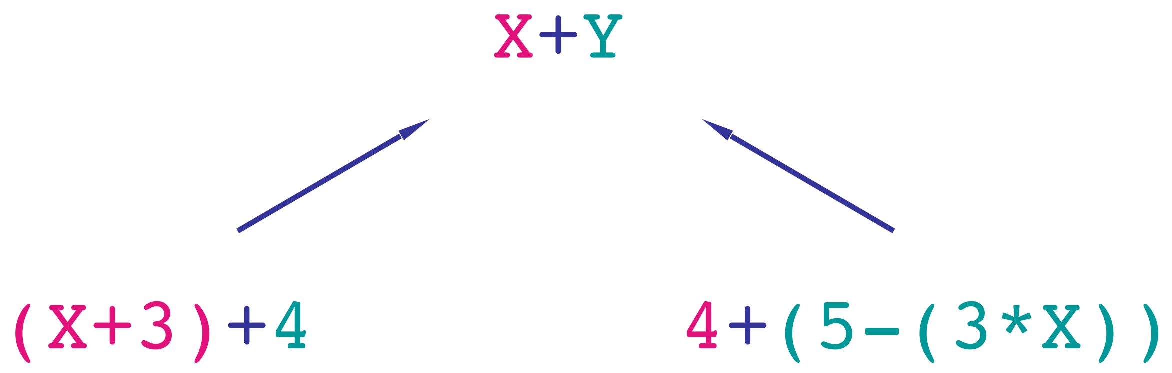
increases the cost of maintenance.

But … it's not always a problem.

What is similar code?

```
(X+3)+4          4+(5-(3*X))
```

What is similar code?



What is similar code?

```
          X+Y

(X+3)+4         4+(5-(3*X))
```

The anti-unification gives the (most specific) common generalisation.

What is similar code?

```
X+Y

(X+3)+4          4+(5-(3*X))

f(Z,W) -> X+Y.
```

The anti-unification gives the (most specific) common generalisation.

What is similar code?

```
X+Y

f(X+3,4)        f(4,5-(3*X))

f(Z,W) -> X+Y.
```

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 = $\min_{i=1..n}(\text{size}(Gen)/\text{size}(E_i))$

What makes a clone (in Erlang)?

Thresholds … and their defaults

Number of expressions ≥ 5

Number of tokens ≥ 20

Number of variables introduced ≤ 4

Similarity = $\min_{i=1..n}(\mathrm{size}(Gen)/\mathrm{size}(E_i))$ ≥ 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 rulests to filter
  RuleSetNameA = "a",
  RuleSetNameB = "b",
  RuleSetNameC = "c",
  RuleSetNameD = "d",
  ... 16 lines which handle the rules sets are elided ...
  %%Remove rulesets
  NewVar_1,
{RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD, FilterKey}.
```

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

```
new_fun(FilterName, FilterKey) ->
  %%Add rulests 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 …

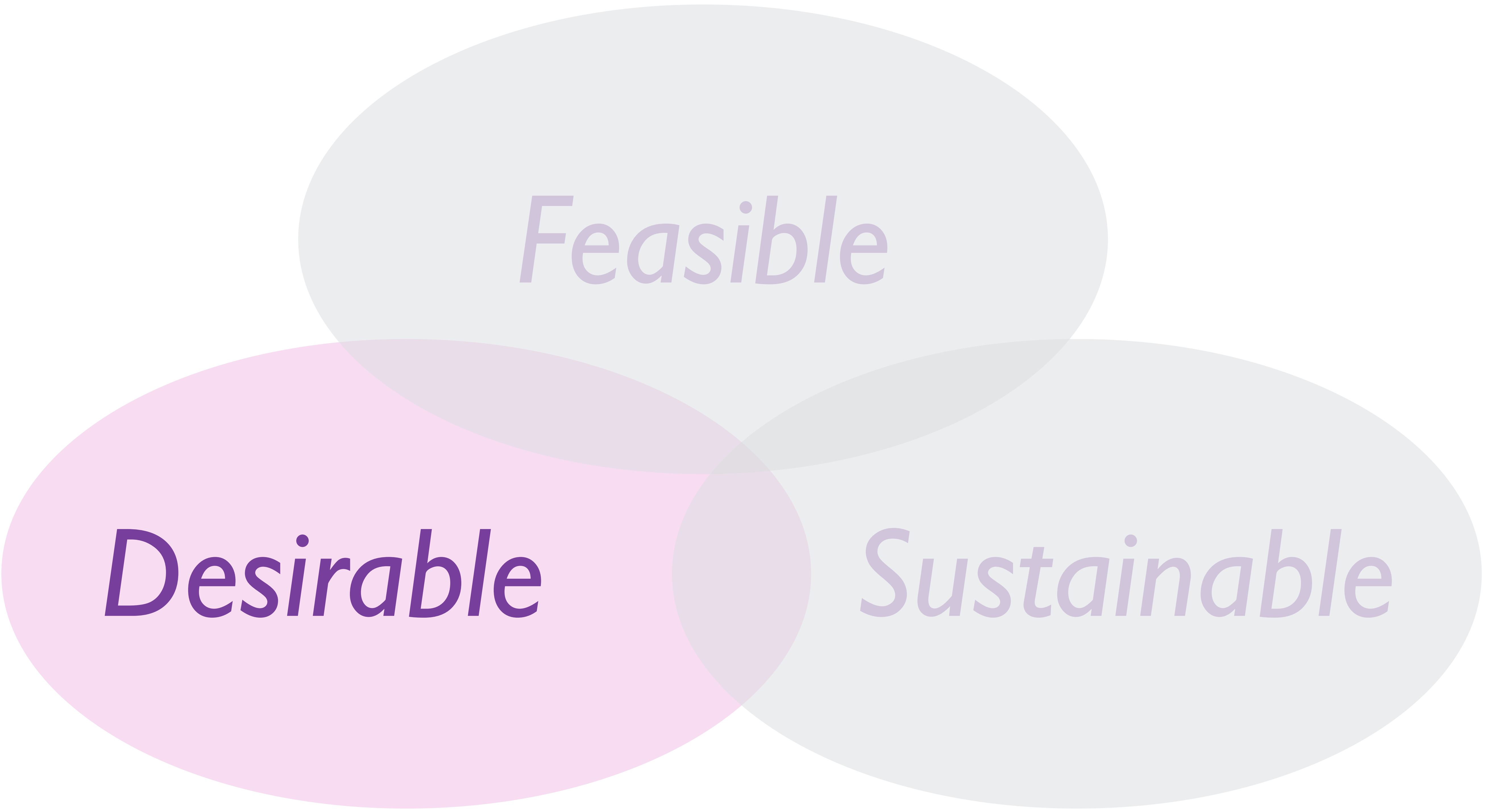
can choose in the right order,

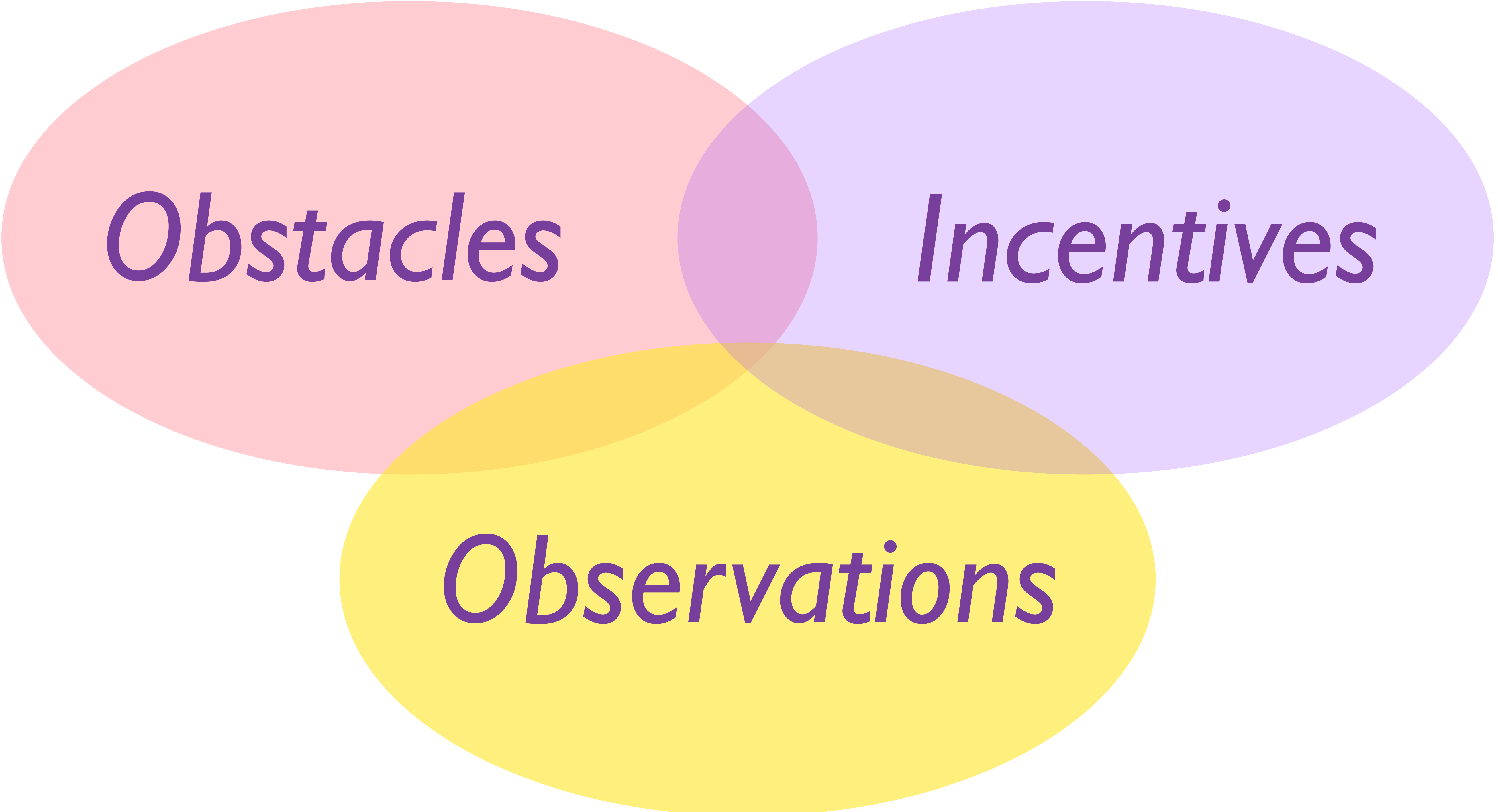
name the clones and their parameters, …

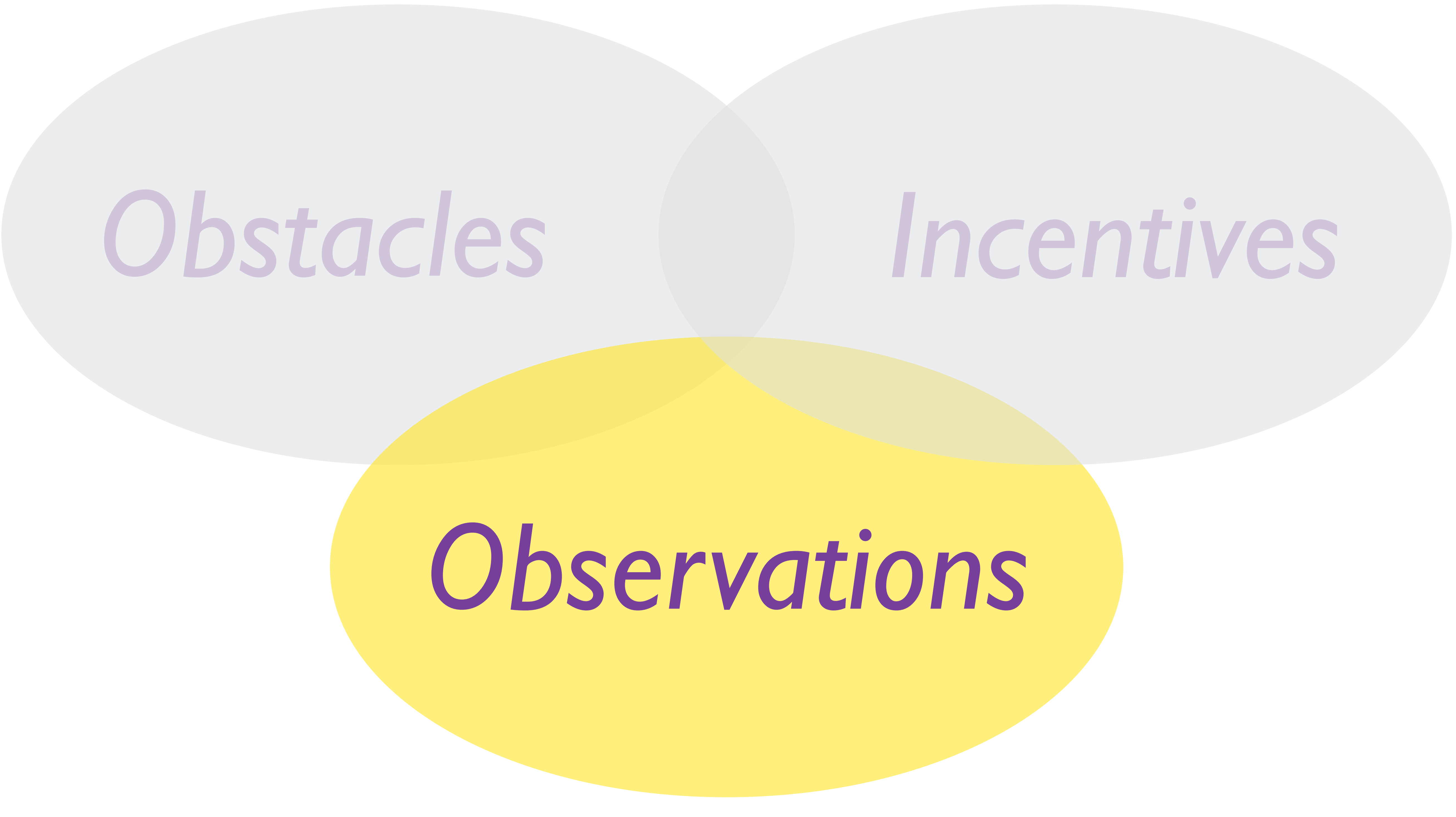
And the domain expert can learn in the process …

e.g. test code example from Ericsson.

Support user involvement rather than full automation.


Feasible
Desirable
Sustainable


Obstacles
Incentives
Observations


Obstacles
Incentives
Observations

User data

Refactoring	Wrangler	LambdaStream
Fold against macro	1	
Fold expression against function	84	17
Generalisation	46	8
Inline variable	3	3
Introduce new variable	22	4
Move function between modules	229	14
Function extraction	119	87
Introduce new macro	1	
Rename function	236	19
Rename module	52	
Rename variable	425	6
Introduce tuple	13	
Unfold function application	12	
Modularity inspection	3	

Keep it simple!

User observations

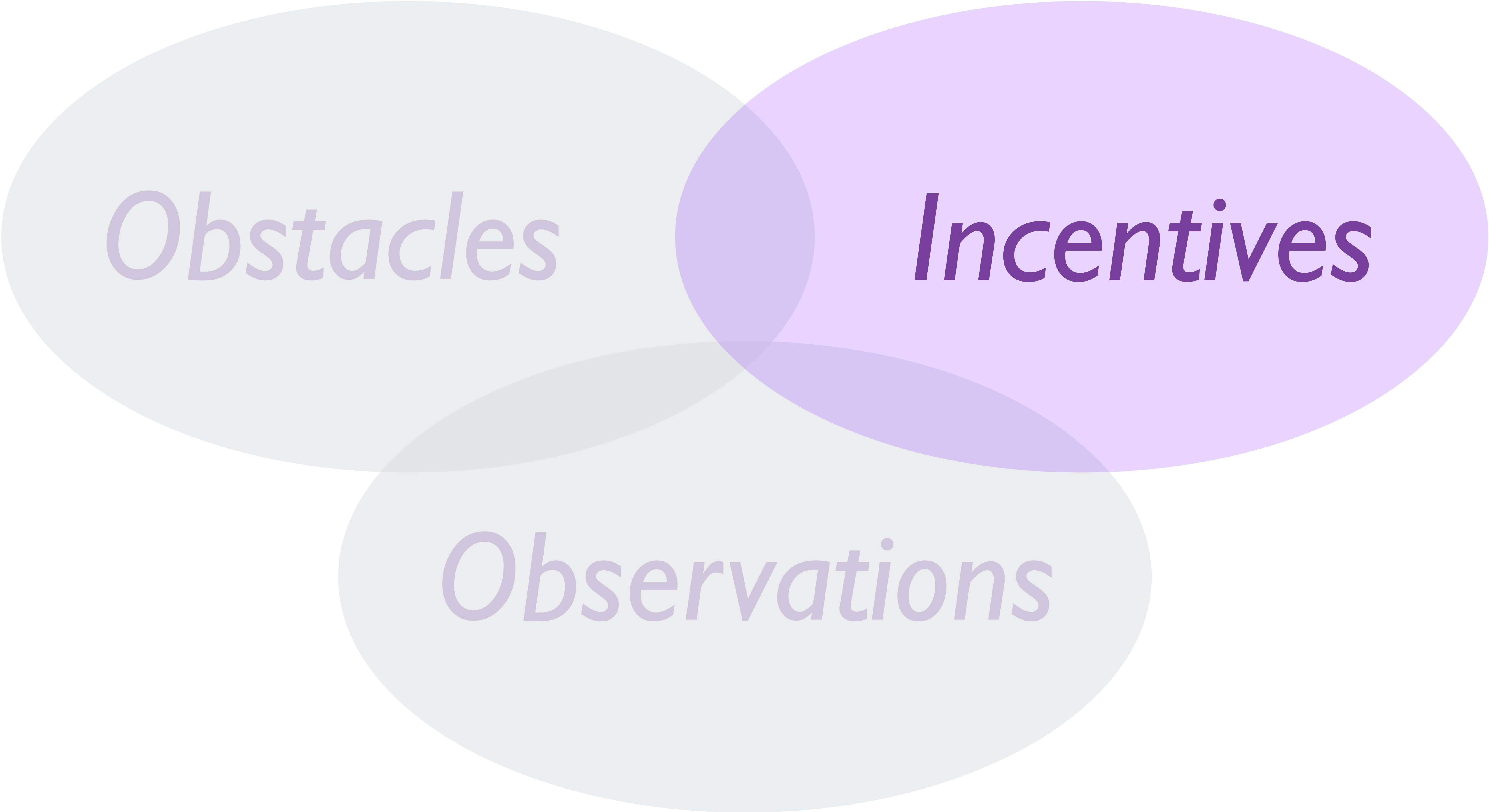
Comprehension exercise on student coursework.

Clone detection exercise with Ericsson staff.

Workflow integration at LambdaStream.

Developing and using DSL with Quviq.

Sitting-in with OCaml group at Jane Street.


Obstacles
Incentives
Observations

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

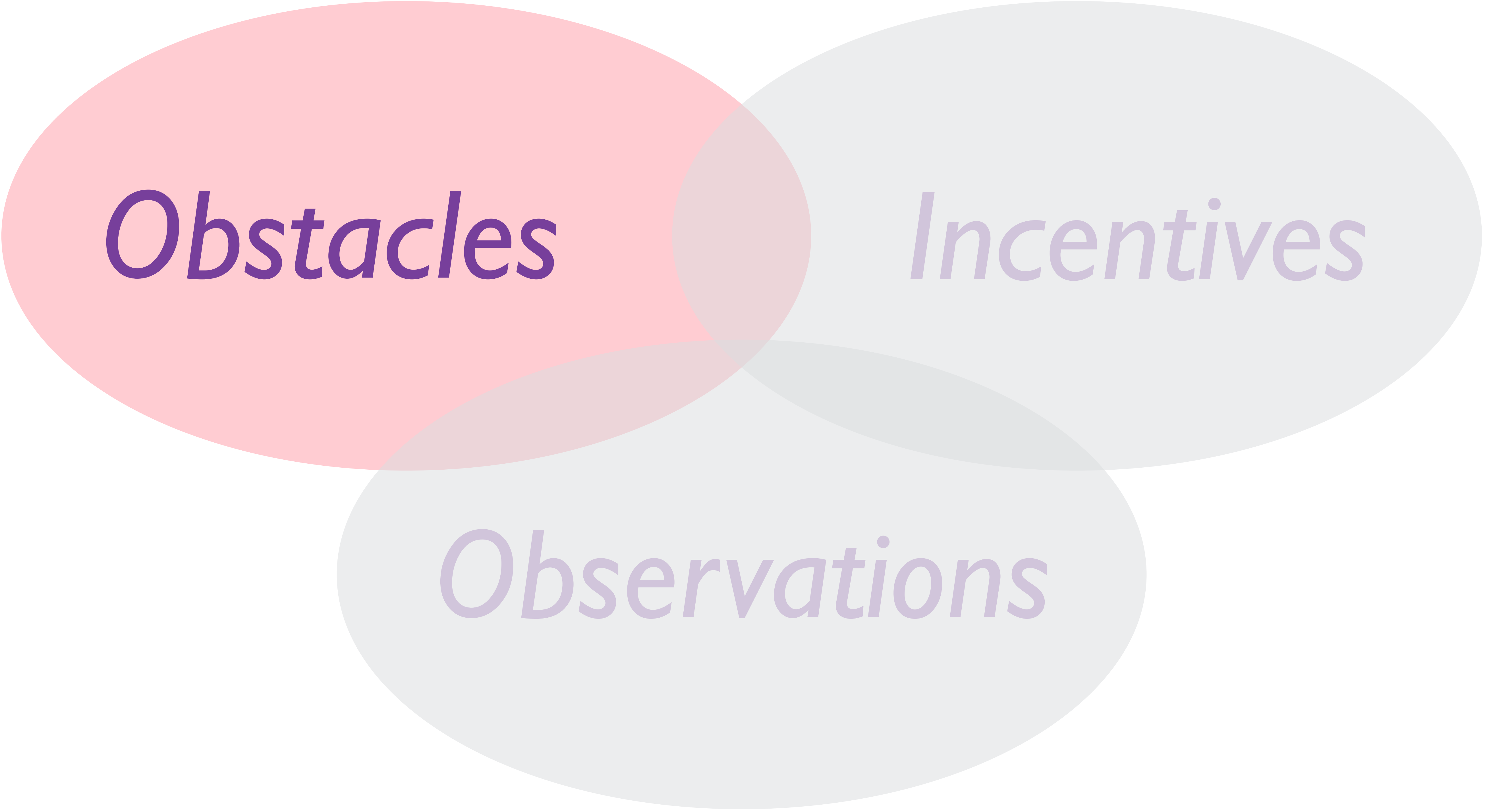
Editor integration … but which are the most popular?

LSP support.

Build and test tools, pre-processors.

Dependencies … and Windows.

Benefits should outweigh costs.


Obstacles
Incentives
Observations

Layout

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
    ]

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

```
{v1, v2, v3}

{v1,v2,v3}
```

Appearance must be right

```
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}

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.

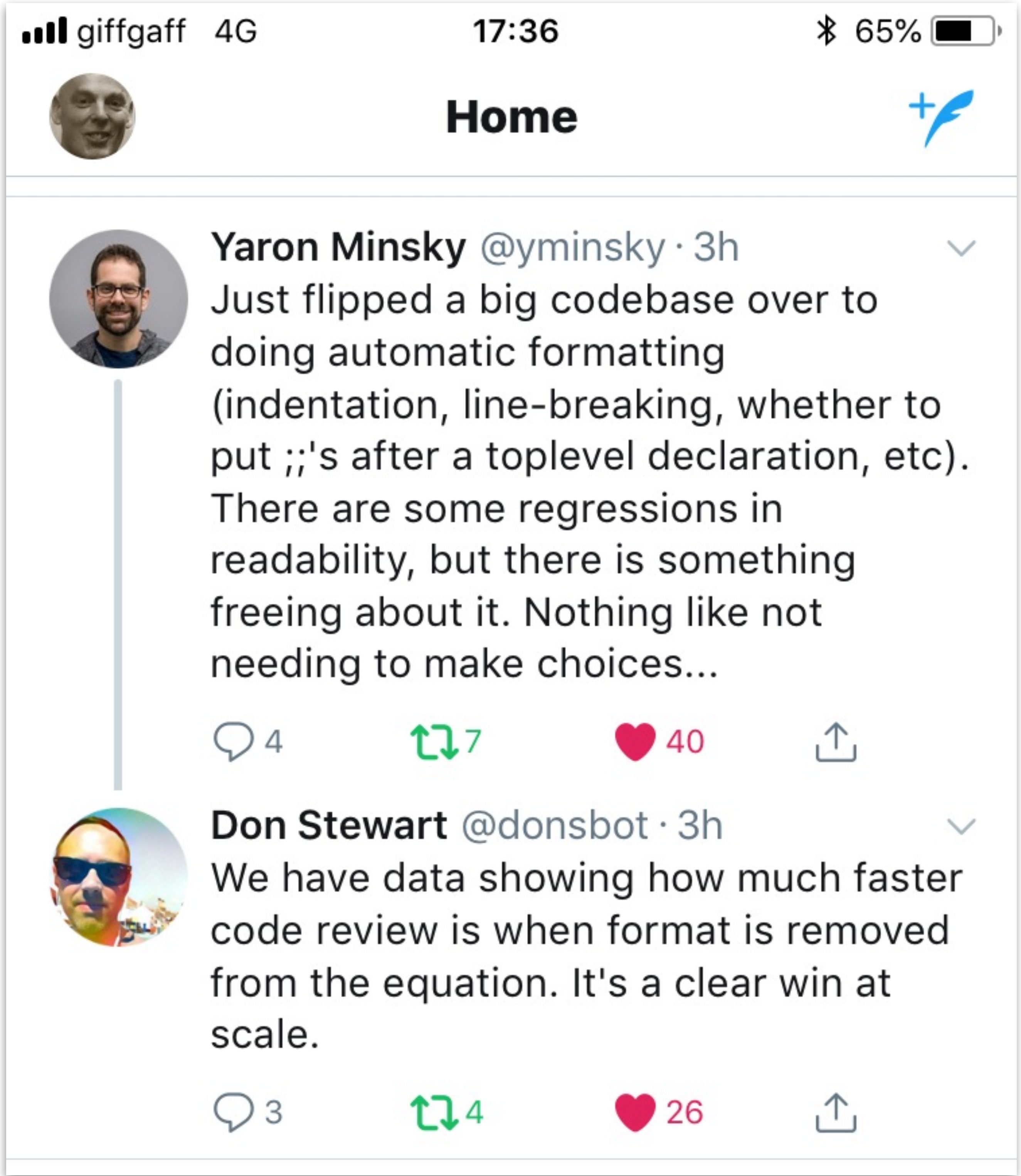
Preserving appearance isn't built in

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*


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

“but there is something freeing about it. Nothing like not needing to make choices …”

I have types … I don’t need a tool

How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Much of what we know about how programmers refactor in the wild is based on studies … *projects. Researchers have* … *these studies in other con*… *tions on which they are ba*… *search on a sound scientifi*… *ing four data sets spannin*… *240 000 tool-assisted refactorings, 2500 developer hours, and 3400 version control commits. Using these data, we cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, we find that programmers frequently* do not *indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, we were able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes. By confirming assumptions and replicating studies made by other researchers, we can have greater confidence that those researchers' conclusions are generalizable.*

a single research method: Weißgerber and Diehl's study of 3 open source projects [18]. Their research method was to apply a tool to the version history of each project to de- … RENAME METHOD and … level refactorings, such … d EXTRACT METHOD, … code changes. One of … y on which refactoring … hanges also took place. What we can learn from this depends on the relative frequency of high-level and mid-to-low-level refactorings. If the latter are scarce, we can infer that refactorings and changes to the projects' functionality are usually interleaved at a fine granularity. However, if mid-to-low-level refactorings are common, then we cannot draw this inference from Weißgerber and Diehl's data alone.

In general, validating conclusions drawn from an individual study involves both replicating the study in wider contexts and exploring factors that previous authors may not have explored. In this paper we use both of these methods to confirm — and cast doubt on — several conclusions that have been published in the refactoring literature.

Up to 90% of refactorings done by hand

ICSE 2009

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

[–] **alan_zimm** **17 points** 1 year ago

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.

perma-link embed save

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.

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

From Monad to Applicative

```
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

```
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
```

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

From `List` to `Vector`

```haskell
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)
```

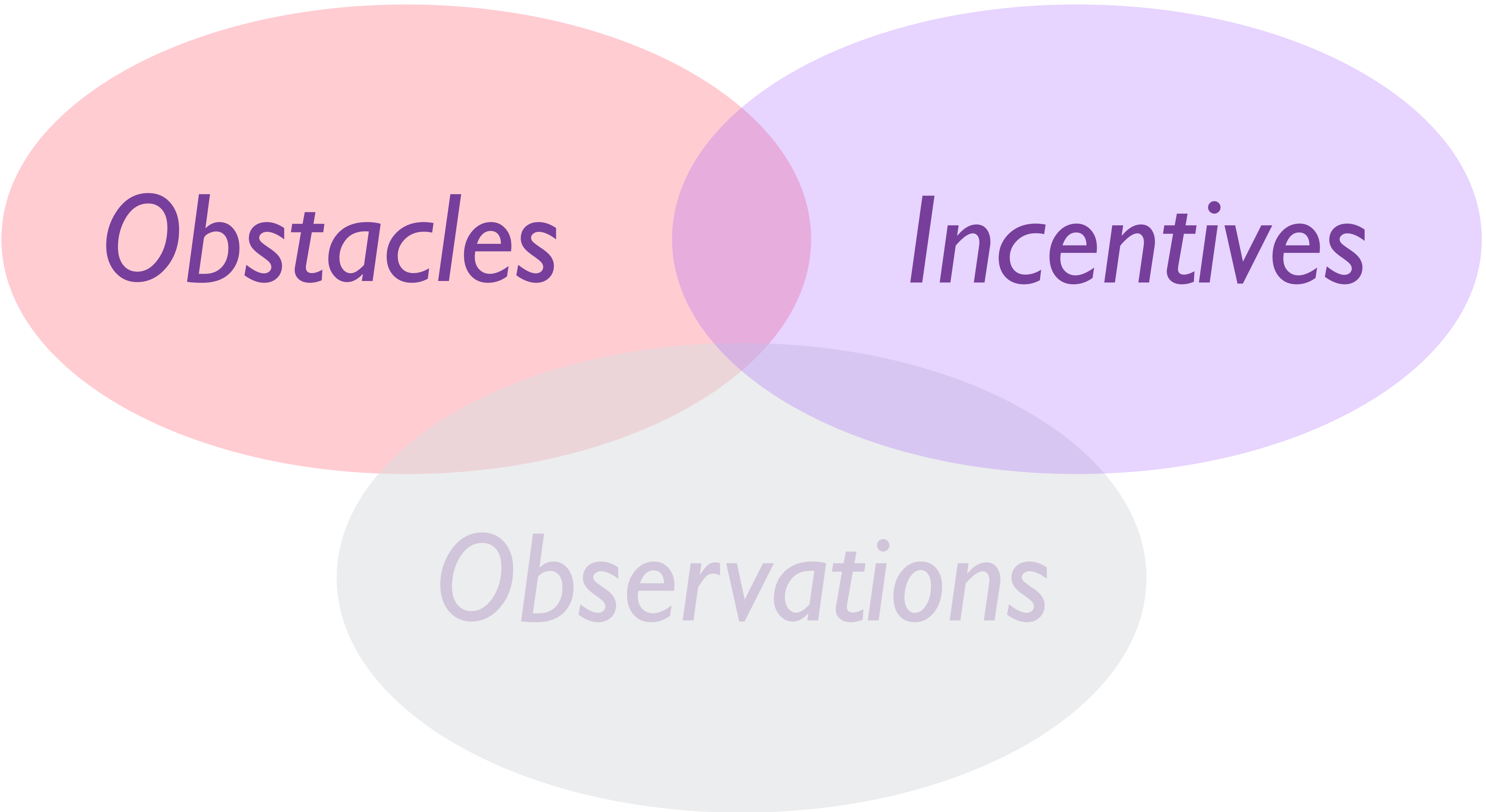
Types vs refactorings?

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)
```

Types can both help and hinder effective refactoring


Obstacles
Incentives
Observations

Why should I trust your refactoring tool on my code?

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

FOCUS: REFACTORING

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.

44 IEEE SOFTWARE | PUBLISHED BY THE IEEE COMPUTER SOCIETY 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

FOCUS: REFACTORING

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.

IEEE Software, Nov/Dec 2015

- Breaking code
- Cannot justify the time spent
- Unpredictable impact
- Difficult to review
- Inadequate tools

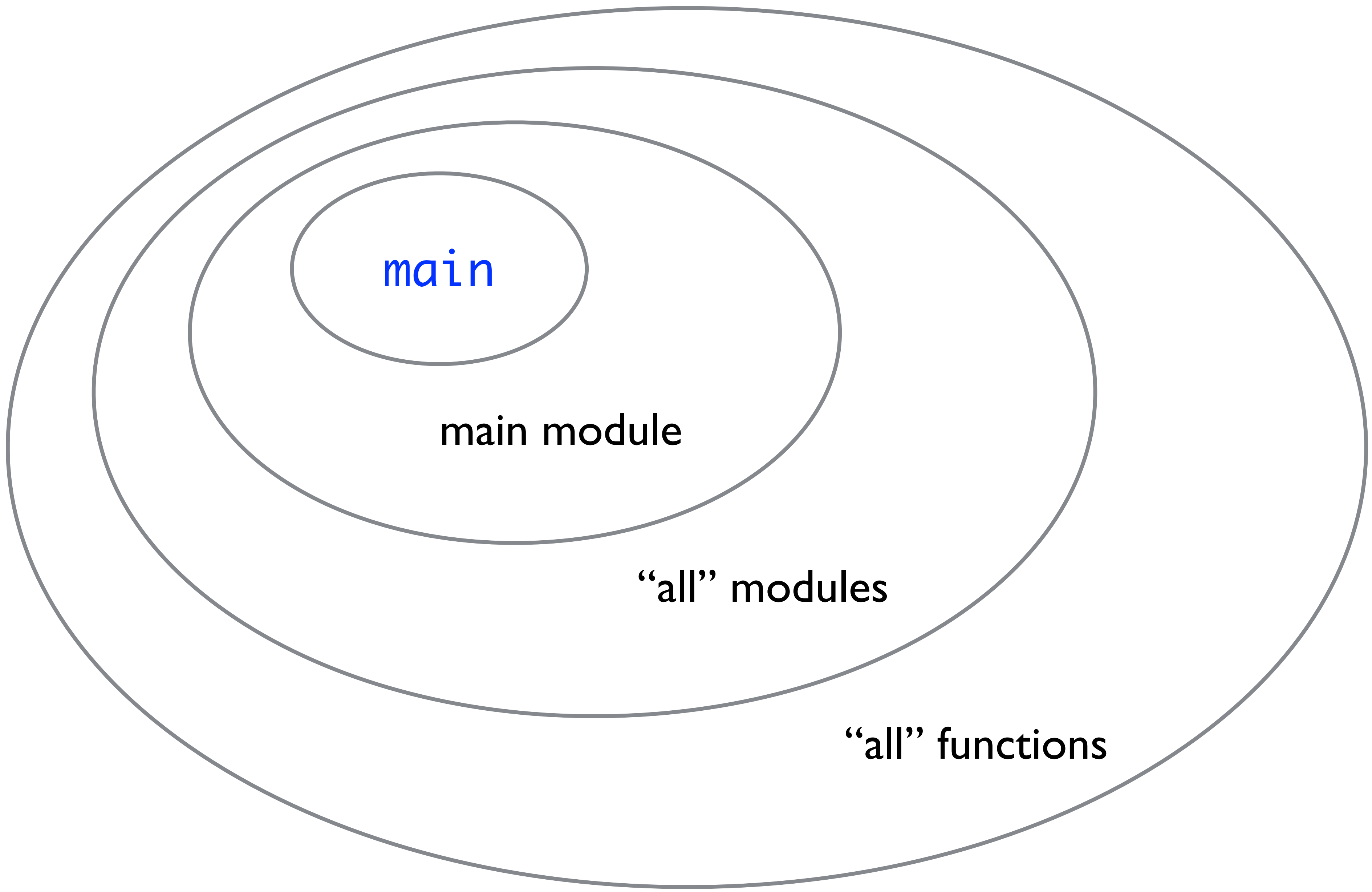
Preserving meaning

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



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

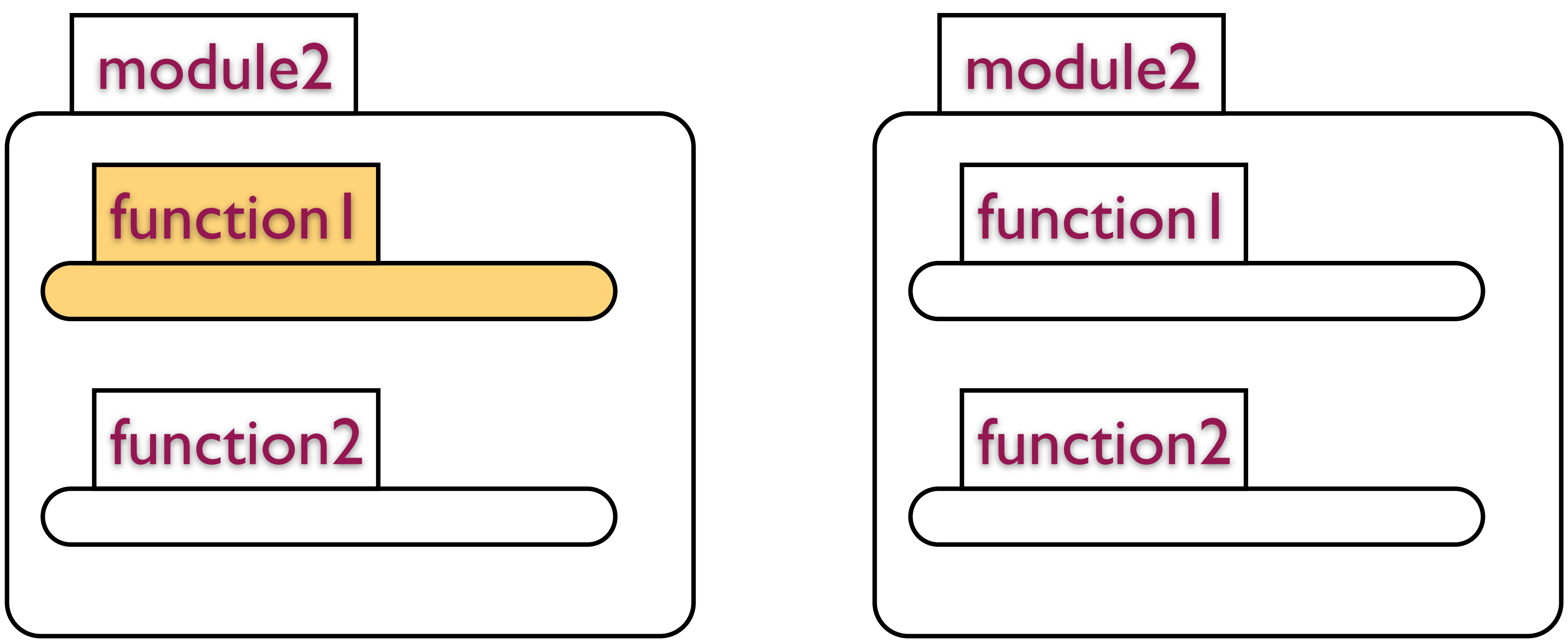
	test	verify
instances of the refactoring	✓	
the refactoring itself		

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).



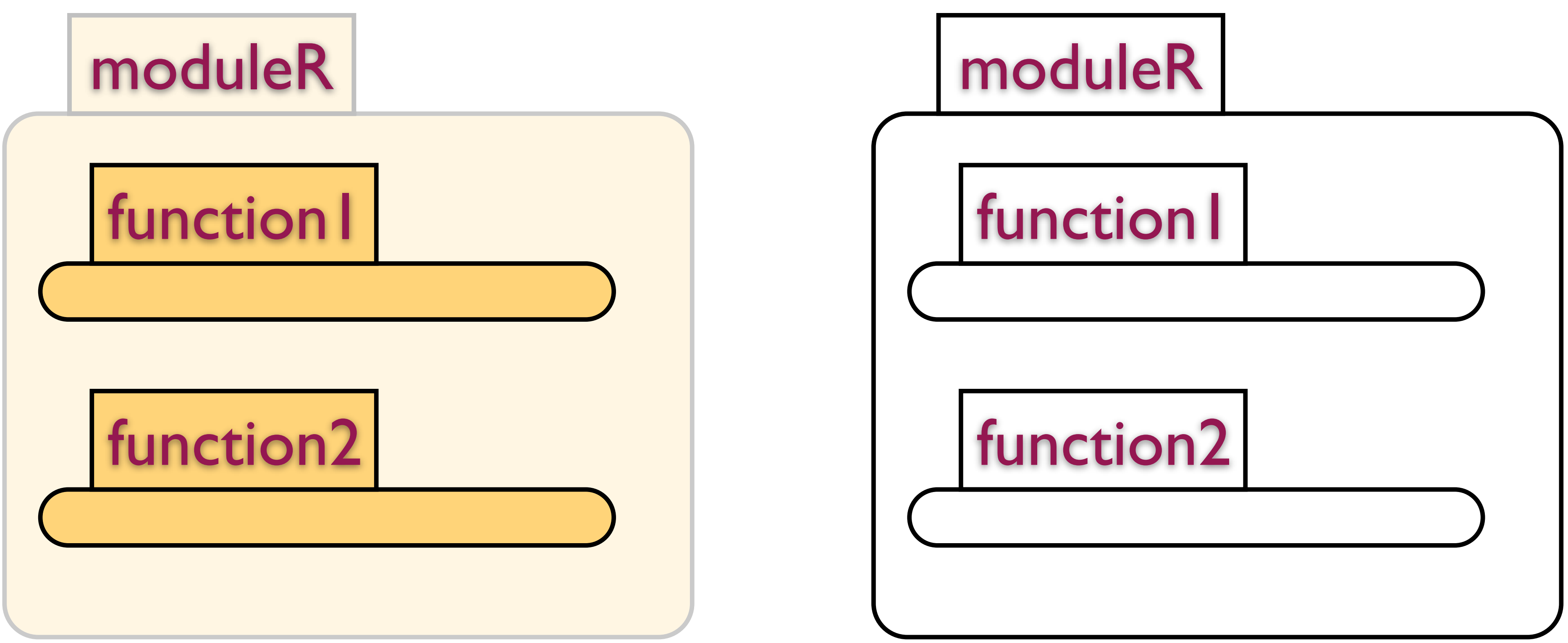
	test	verify
instances of the refactoring		
the refactoring itself	✓	

Fully random

Generate random modules,

… generate random refactoring commands,

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



Verification

	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.

Shallow embedding

	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

```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
```

```
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
```

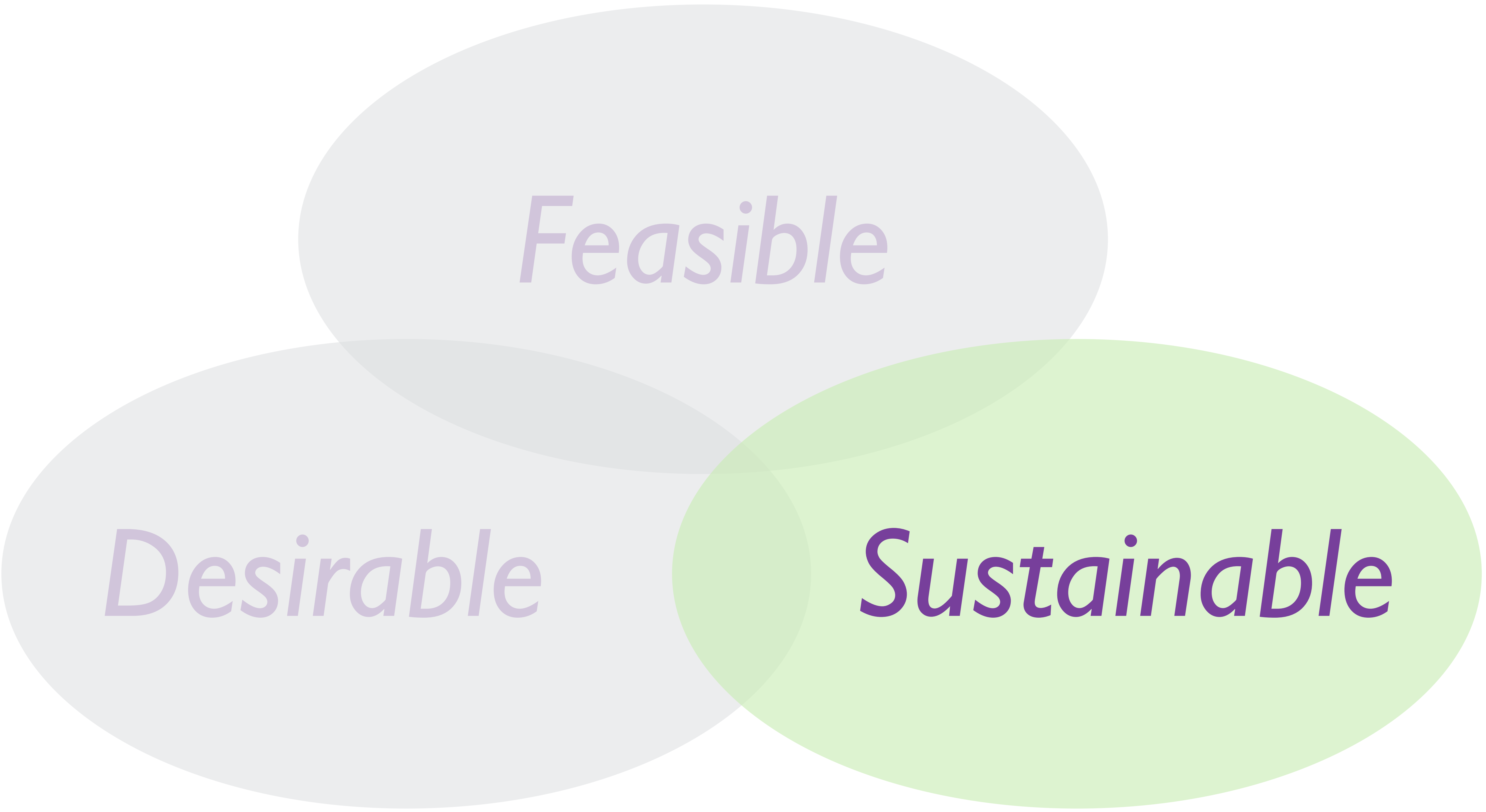
```
h' :: Integer->Integer->Integer

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

k :: Integer->Integer

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

```
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	✓	✓

Trust is a complicated, multi-dimensional issue ... but we're working on it.



dschool.stanford.edu

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.

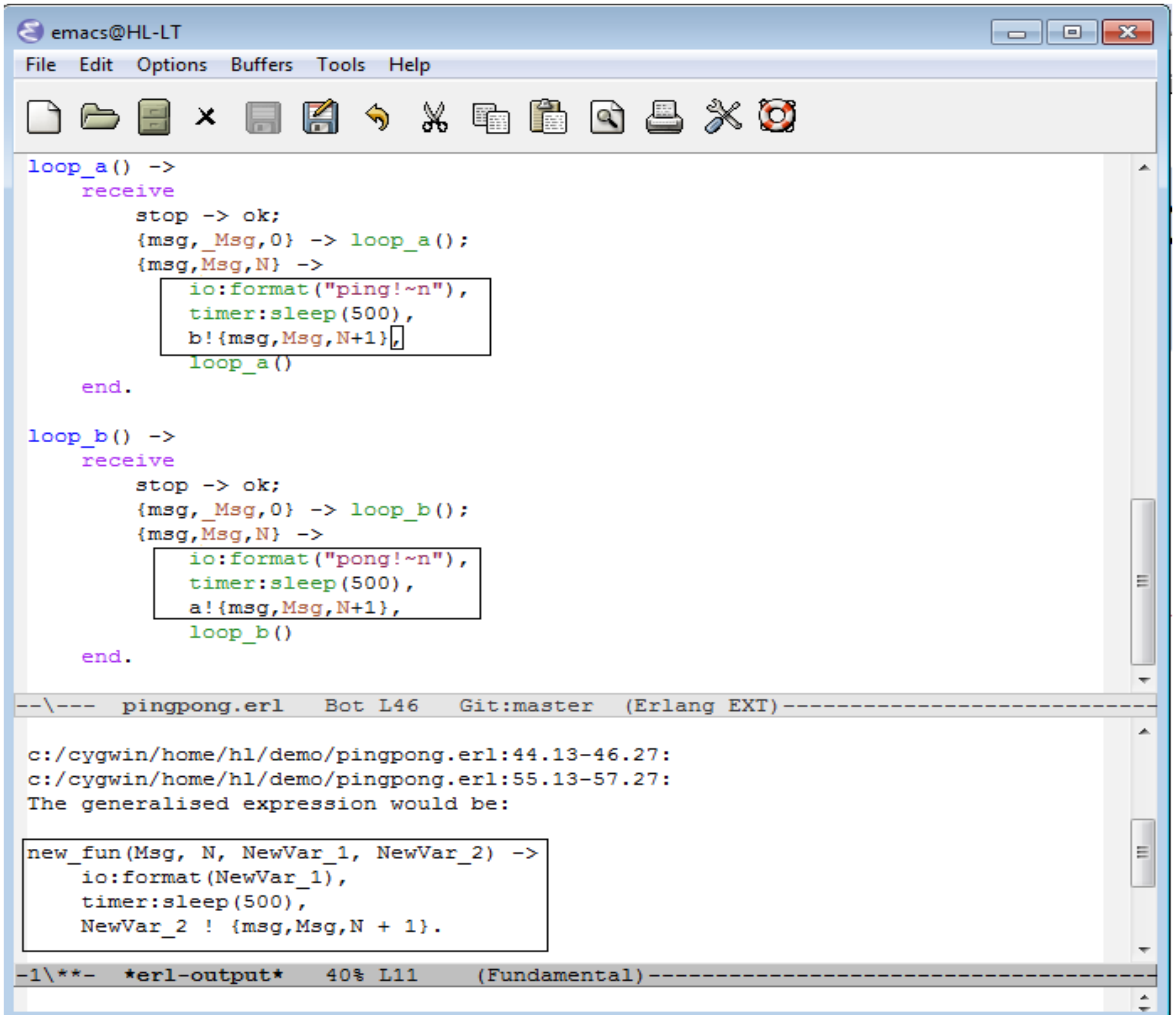
API: templates and rules … in Erlang

```
?RULE(Template, NewCode, Cond)
```

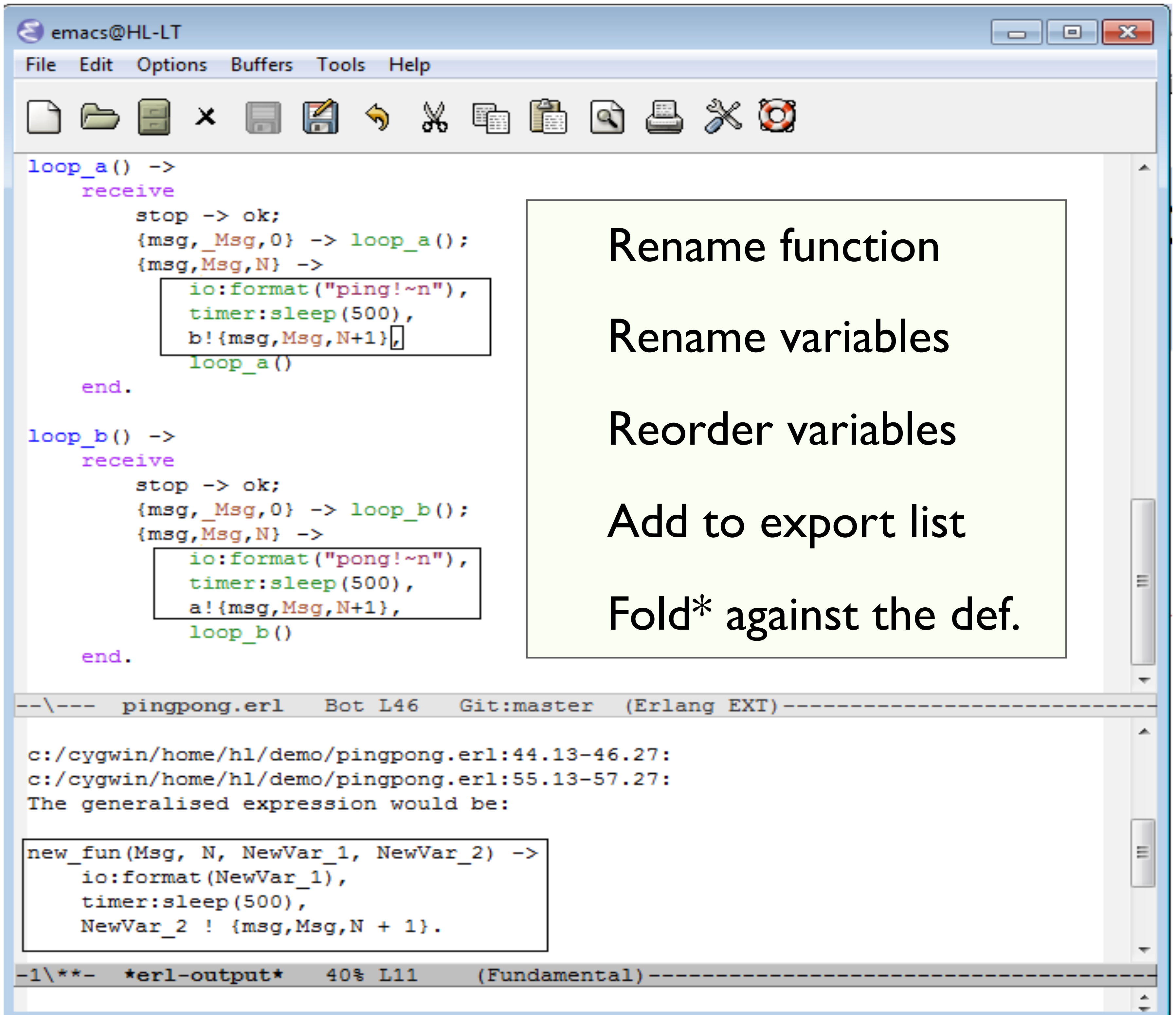
The old code, the new code and the pre-condition.

```
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



Clone removal

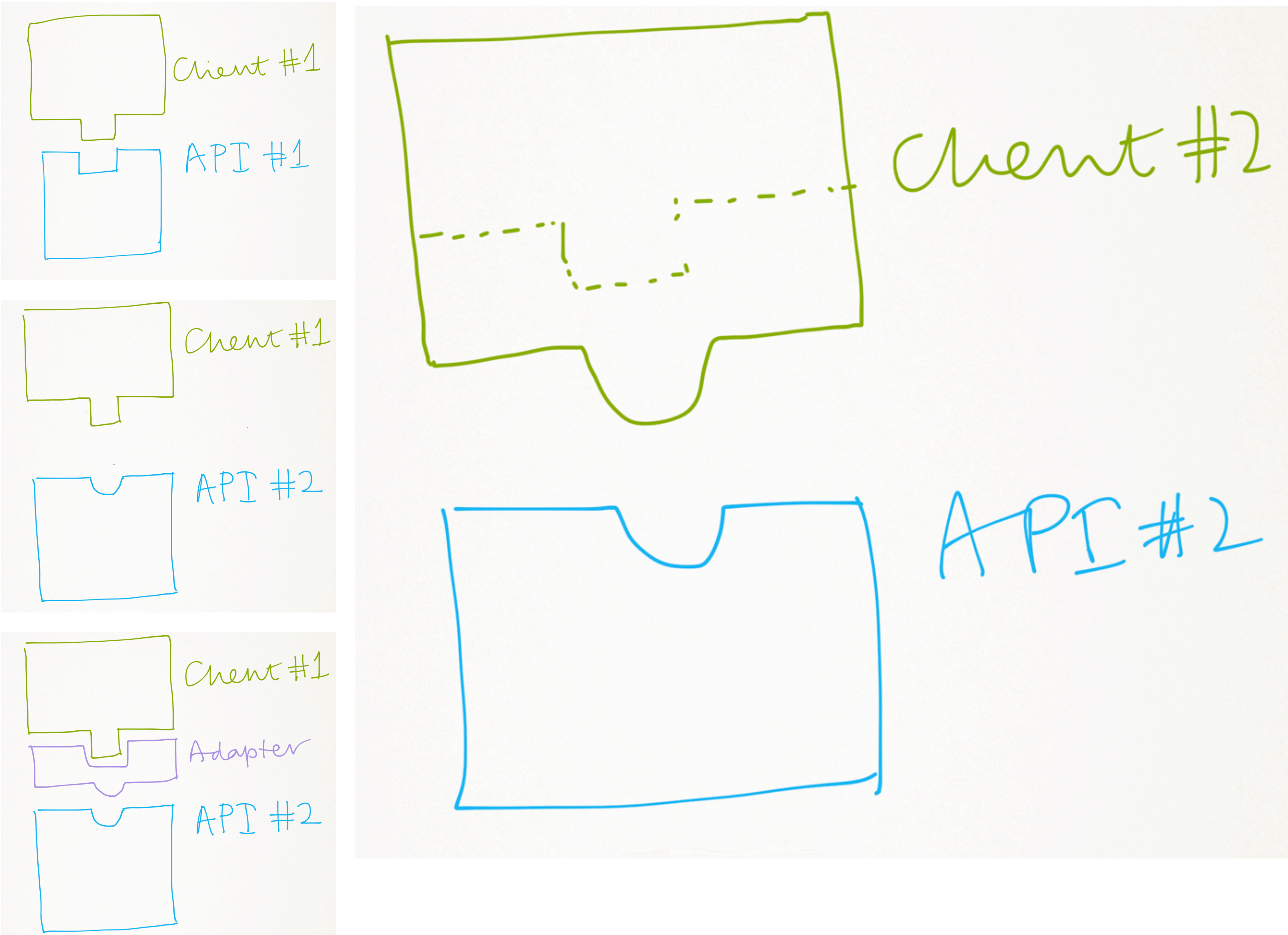


Clone removal in the DSL

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 )
     ]).
```


Client #1
API #1
Client #1
API #2
Client #1
Adapter
API #2
Client #2
API #2

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.

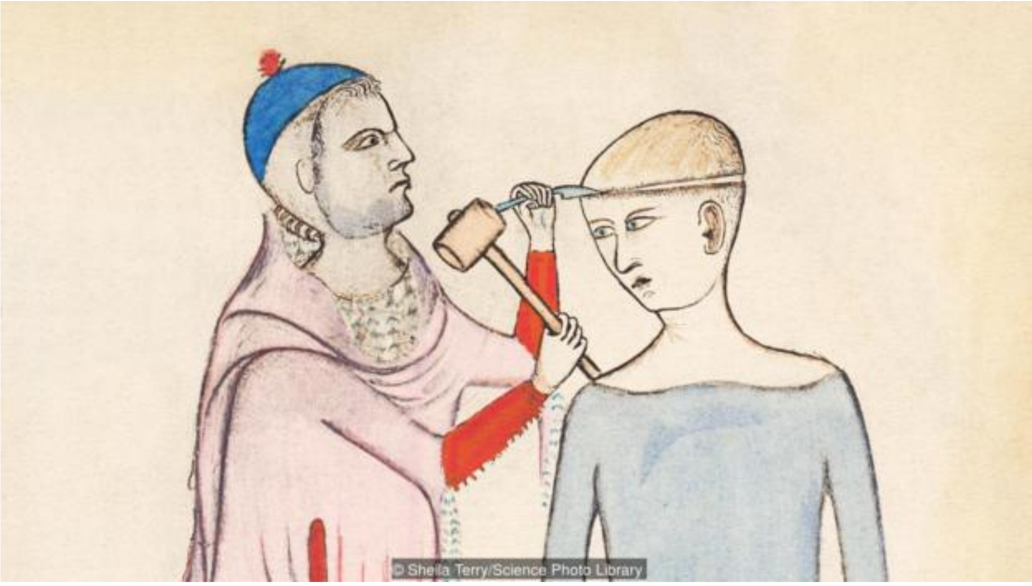
No layout choice: you have to conform to layout rules.

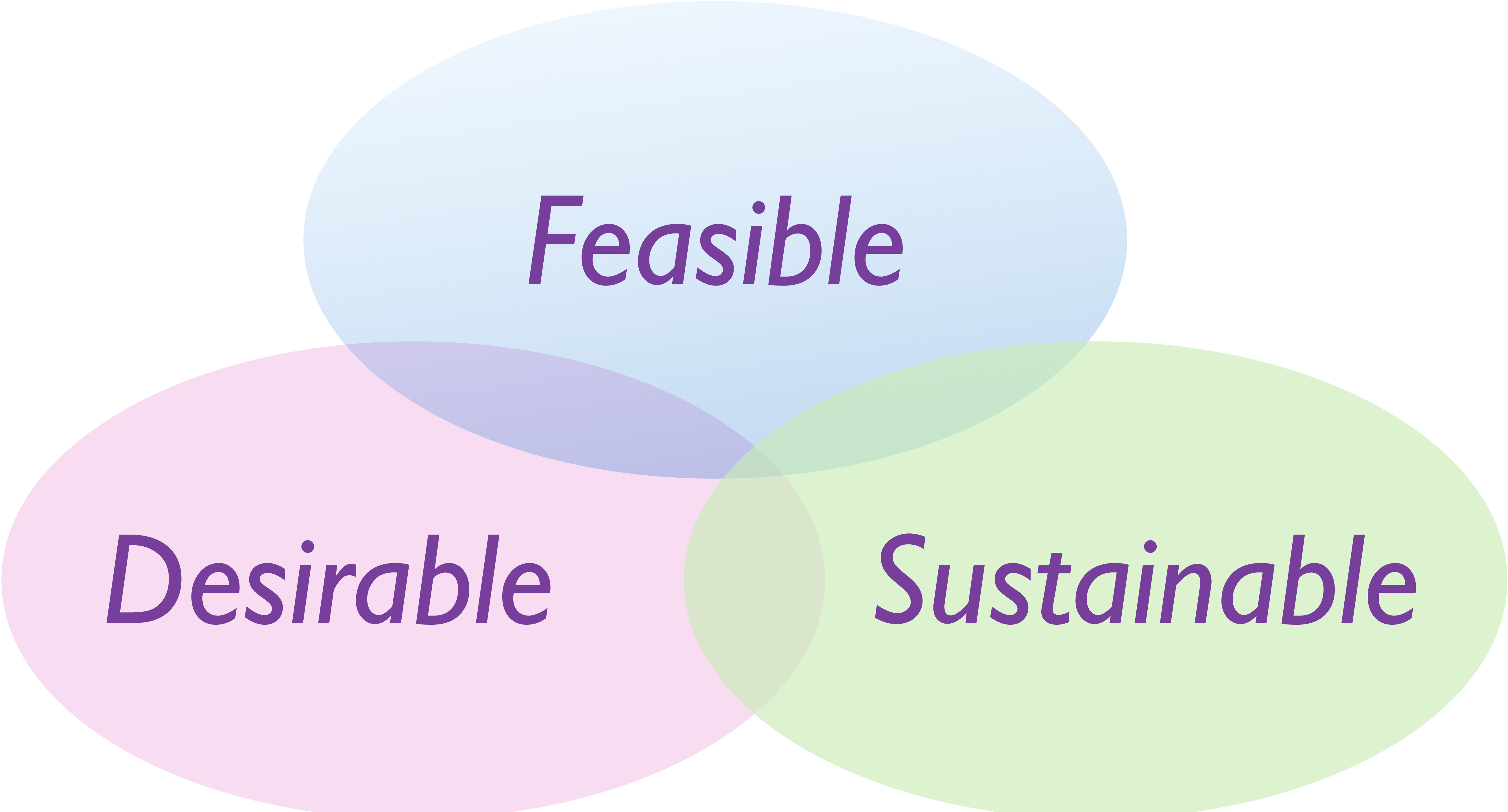
No macros, reflection, …

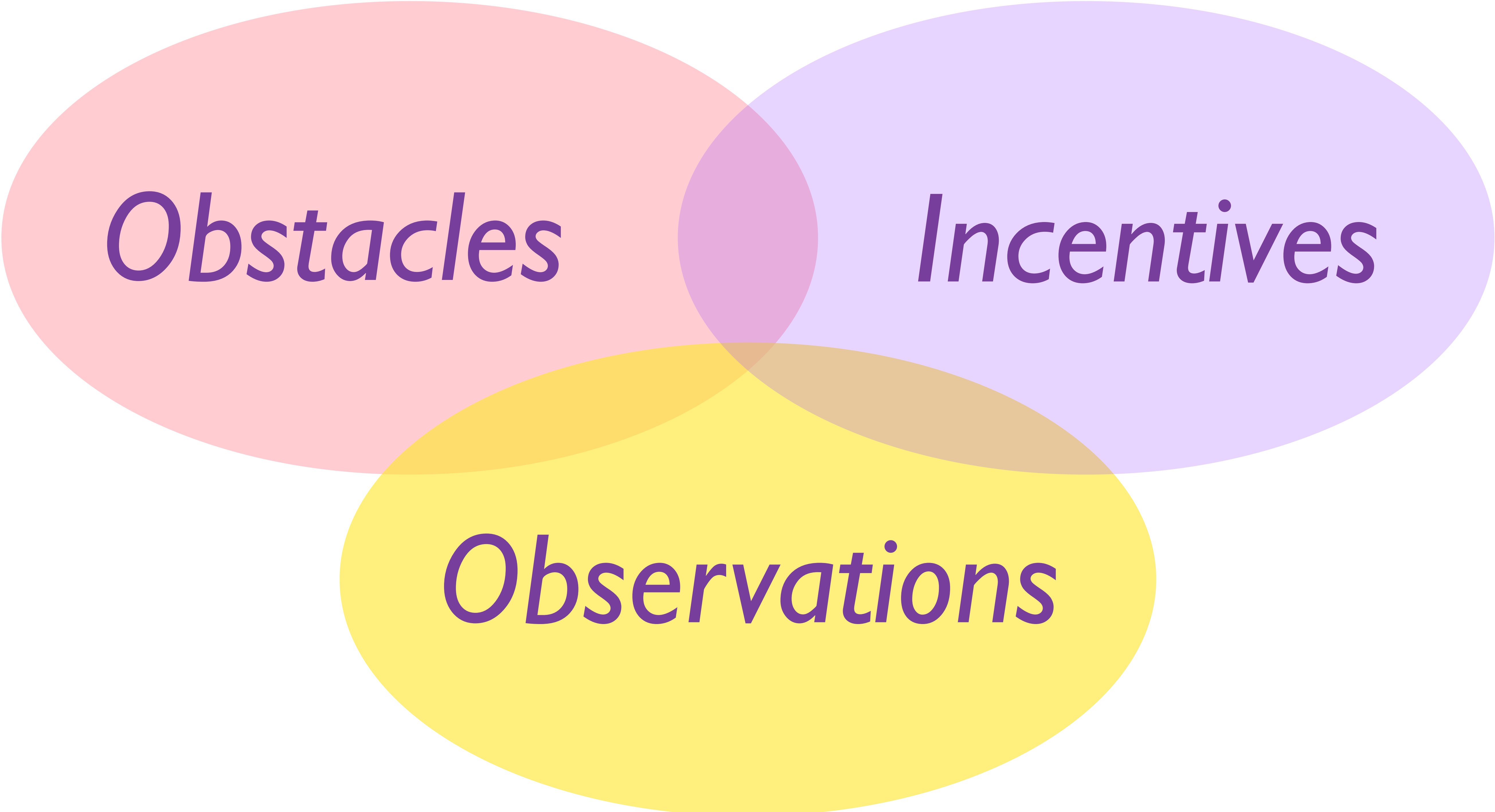
Compiler stability

Integration with a semantically-aware change management tool.

Theory of patches, …




Feasible
Desirable
Sustainable


Obstacles
Incentives
Observations

https://github.com/alanz/HaRe

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

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