Building trustworthy refactoring tools

Simon Thompson, University of Kent, UK

Joint work with Thomas Arts, Dániel Drienyovszky,
Dániel Horpácsi, Huiqing Li and Nik Sultana

Why should I trust my code to your refactoring tool?

Outline

What we do … and how we do it

Psychological, pragmatic and technical

A range of equivalences

Testing … property-based testing

Verification … manual and automated

System-level and program-level

Refactoring

Change ***how*** a program works …

… without changing ***what*** it does.

Why refactor?

Extension and reuse

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       io:format("ping!~n"),
       timer:sleep(500),
       b ! {msg, Msg, N - 1},
       loop_a()
     end.
```

Let’s turn this into a function

Why refactor?

Extension and reuse

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1},
        loop_a()
    end.
```

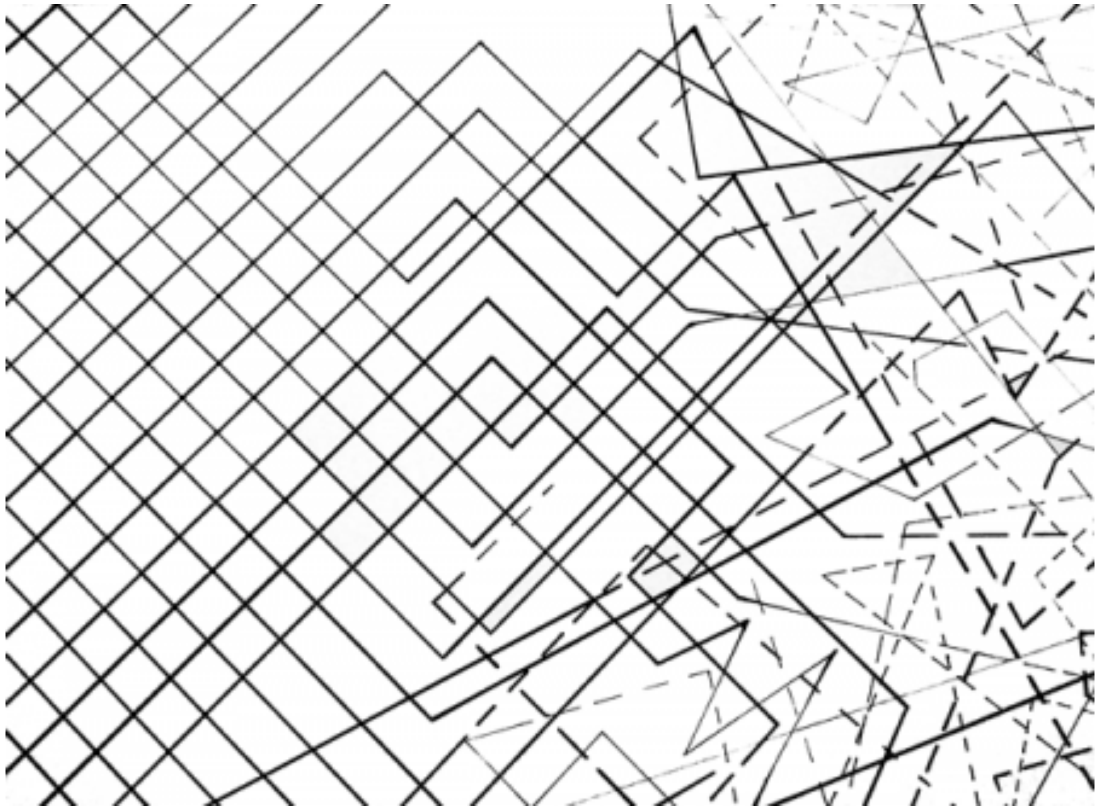
```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
     end.

body(Msg,N) ->
       io:format("ping!~n"),
       timer:sleep(500),
       b ! {msg, Msg, N - 1}.
```

Why refactor?

Counteract decay ... comprehension

"Clones considered harmful": detect and eliminate duplicate code.

Improve the module structure: remove loops, for example.



How to refactor?

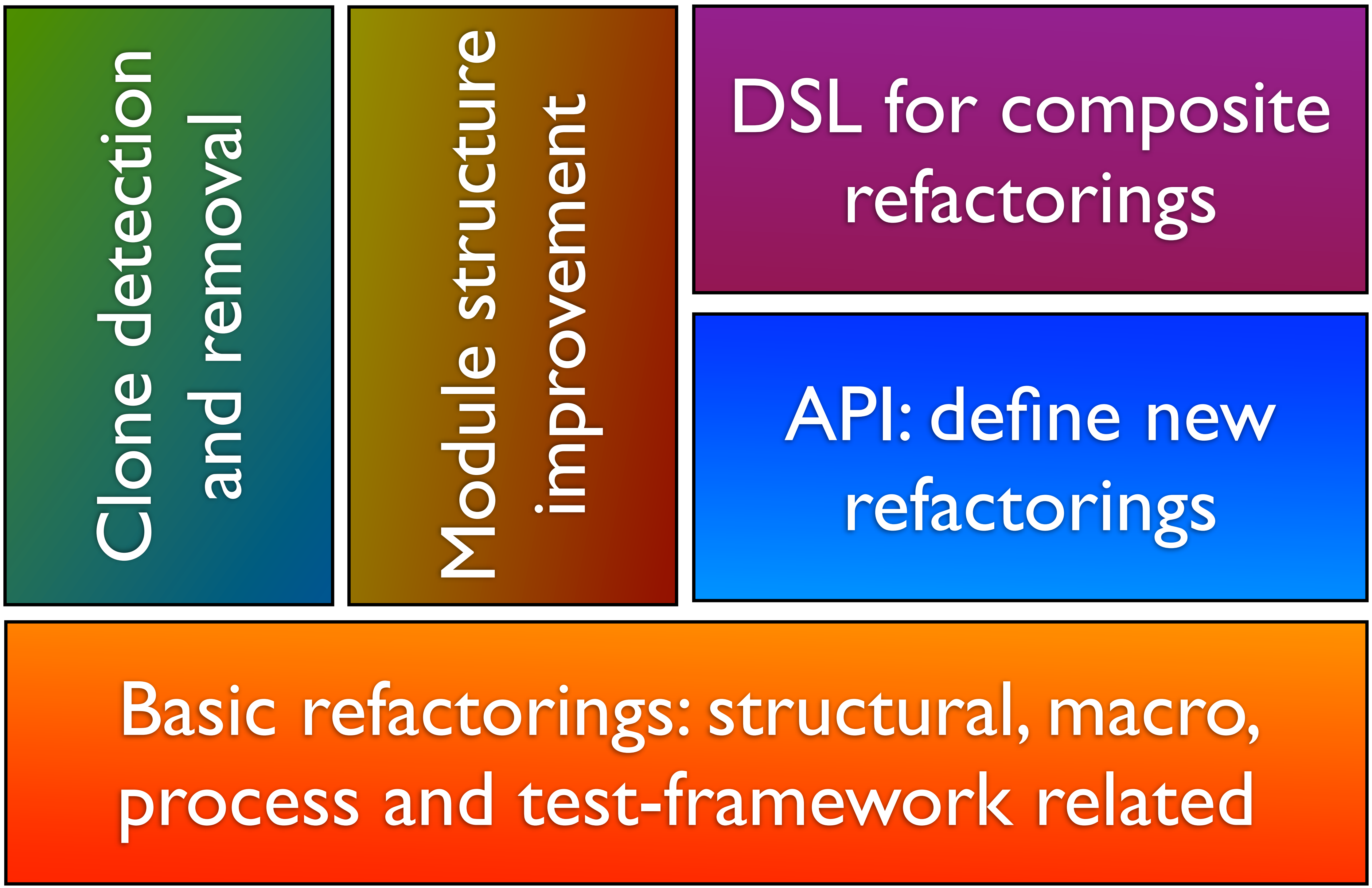
By hand … using an editor.

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

Tool-supported.

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

Wrangler


Clone detection and removal
Module structure improvement
DSL for composite refactorings
API: define new refactorings
Basic refactorings: structural, macro, process and test-framework related

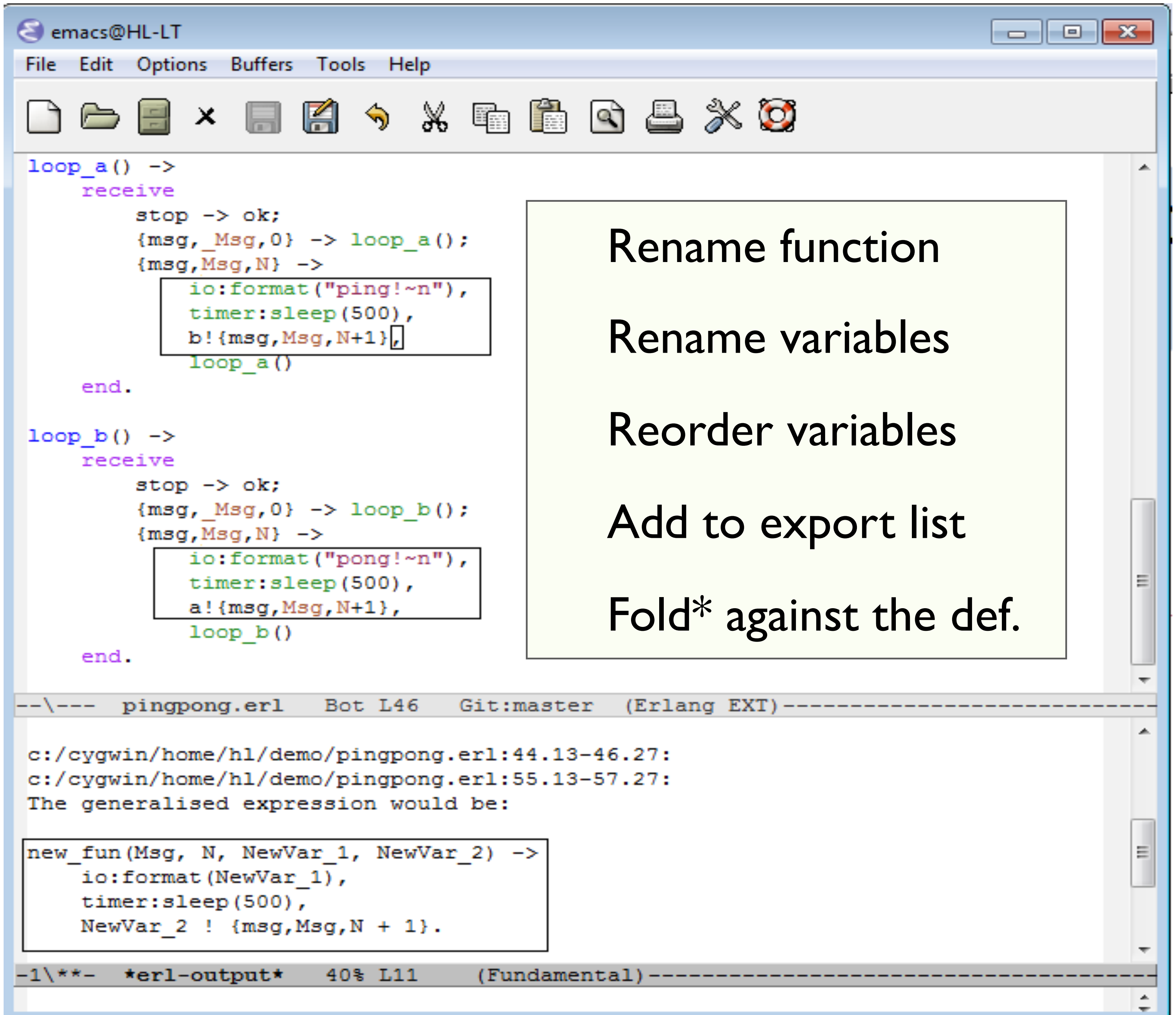
API: templates and rules … in Erlang

```
?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 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 )
     ]).
```

Wrangler in a nutshell

Automate the simple things, and …

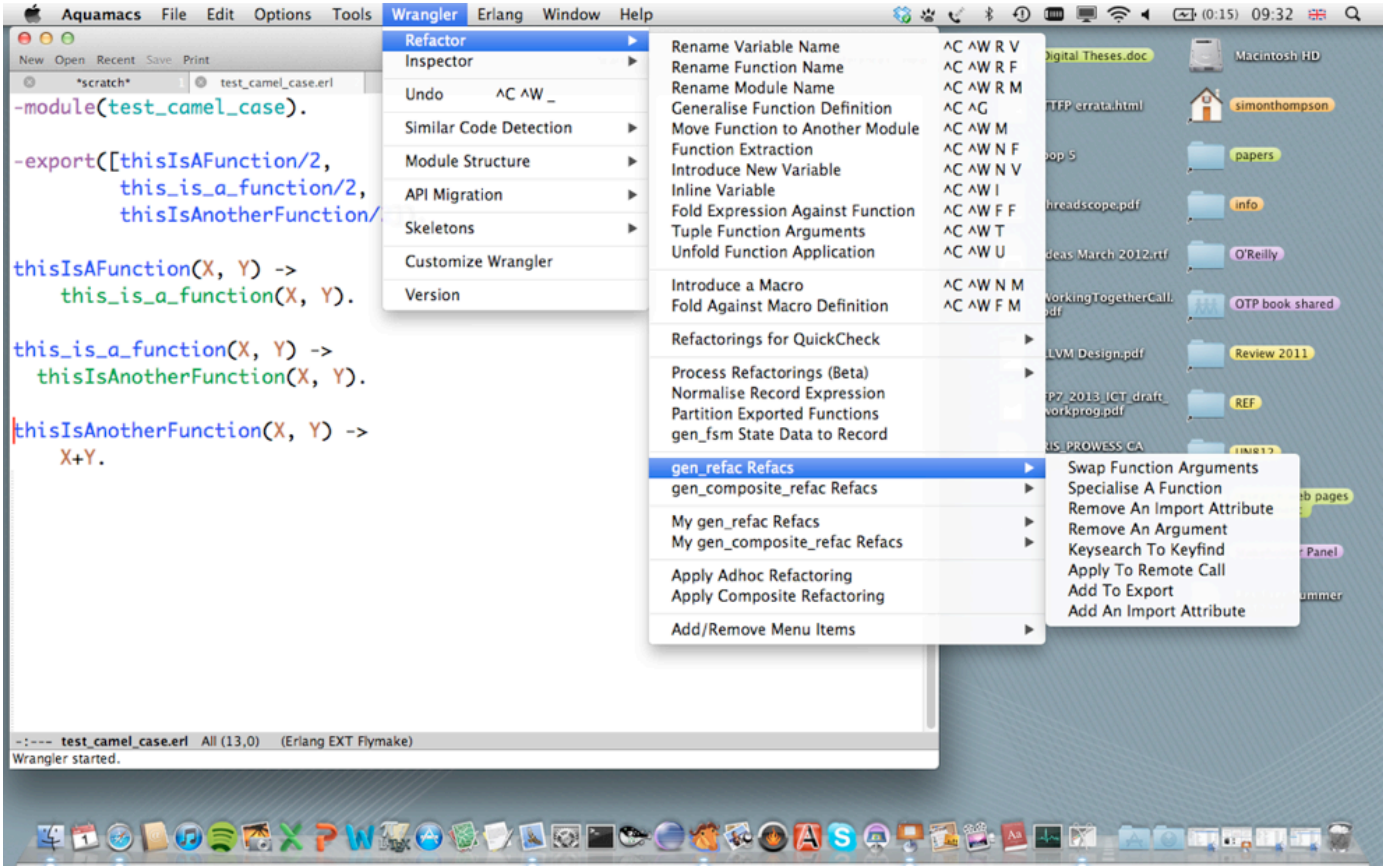
… provide decision support tools otherwise.

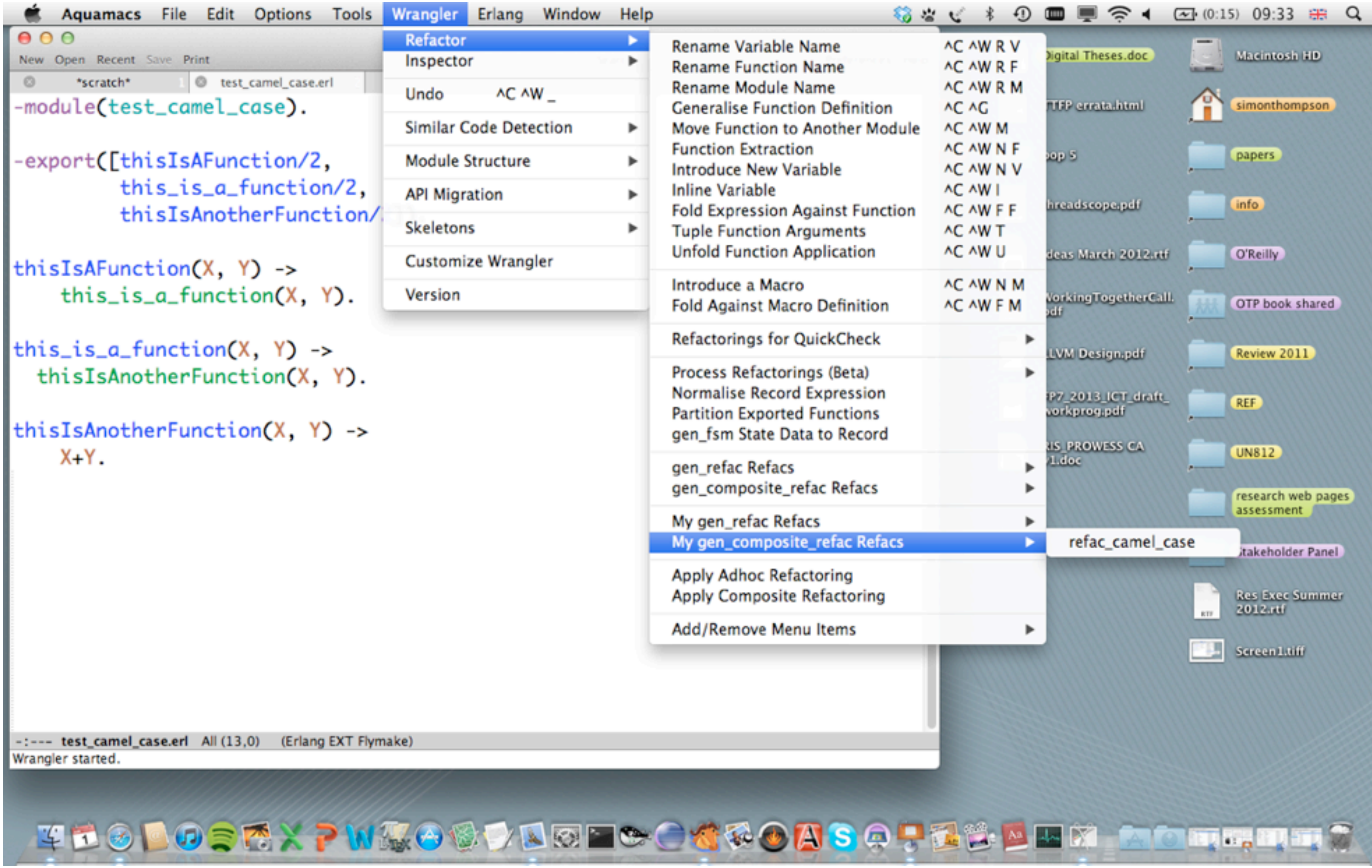
Embed in common IDEs: emacs, eclipse, …

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

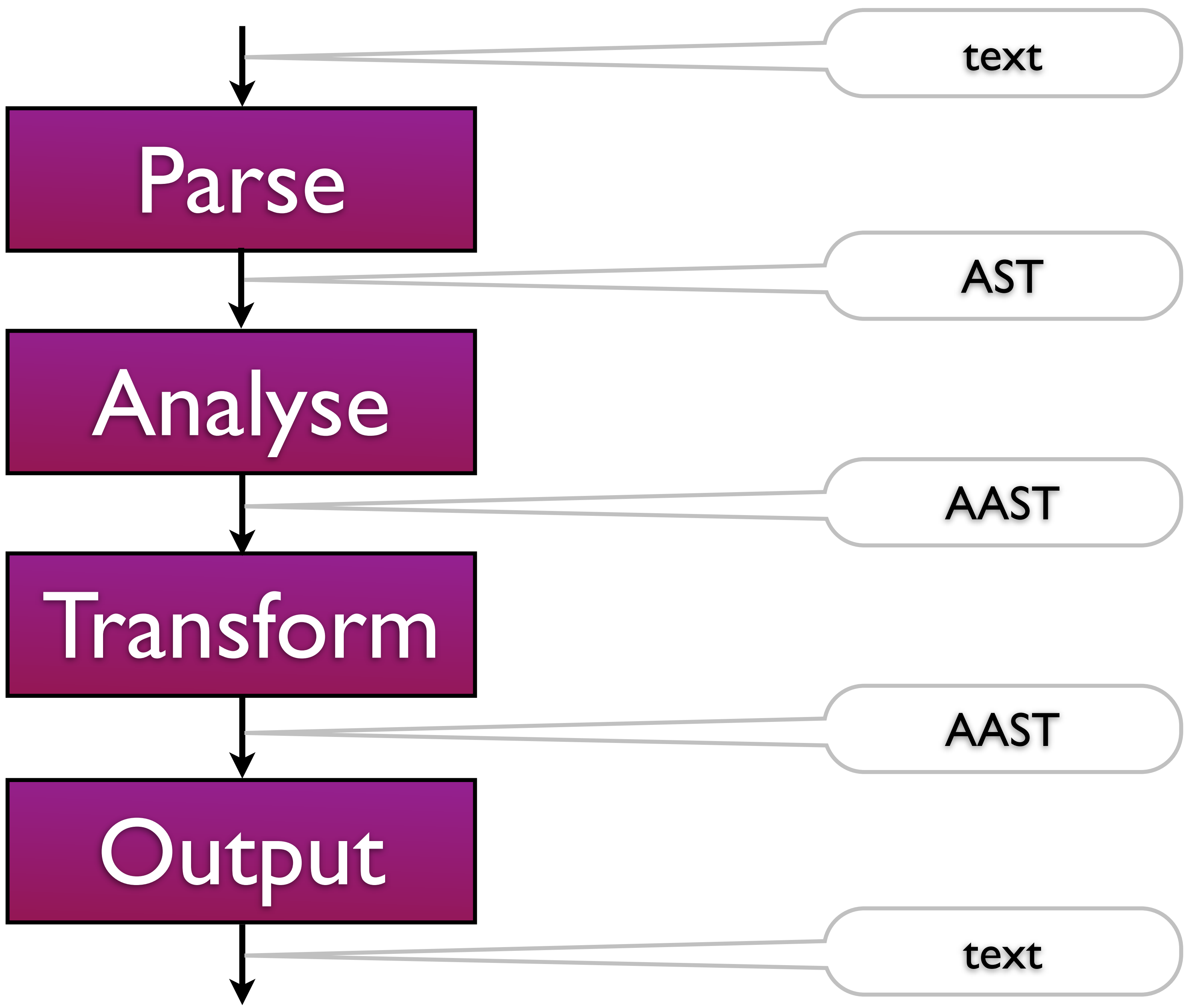
Faithful to layout and comments.

Build in Erlang and apply the tool to itself.


Aquamacs File Edit Options Tools Wrangler Erlang Window Help
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


Aquamacs File Edit Options Tools Wrangler Erlang Window Help
09:33
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
refac_camel_case
Apply Adhoc Refactoring
Apply Composite Refactoring
Add/Remove Menu Items
Macintosh HD
simonthompson
papers
info
O'Reilly
OTP book shared
Review 2011
REF
UN812
research web pages assessment
Res Exec Summer 2012.rtf
Screen1.tiff

Under the hood



Why should I trust my code to your refactoring tool?

Psychological and social issues

Open Source … confidence in the code … other committers.

The benefits outweigh the risk / cost …

… might even be OK to introduce some faults?

Openness of the system …

… you can check the changes that a refactoring makes,

… and for the DSL can see which refactorings performed.

Benefit ≫ risk: removing bug preconditions

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

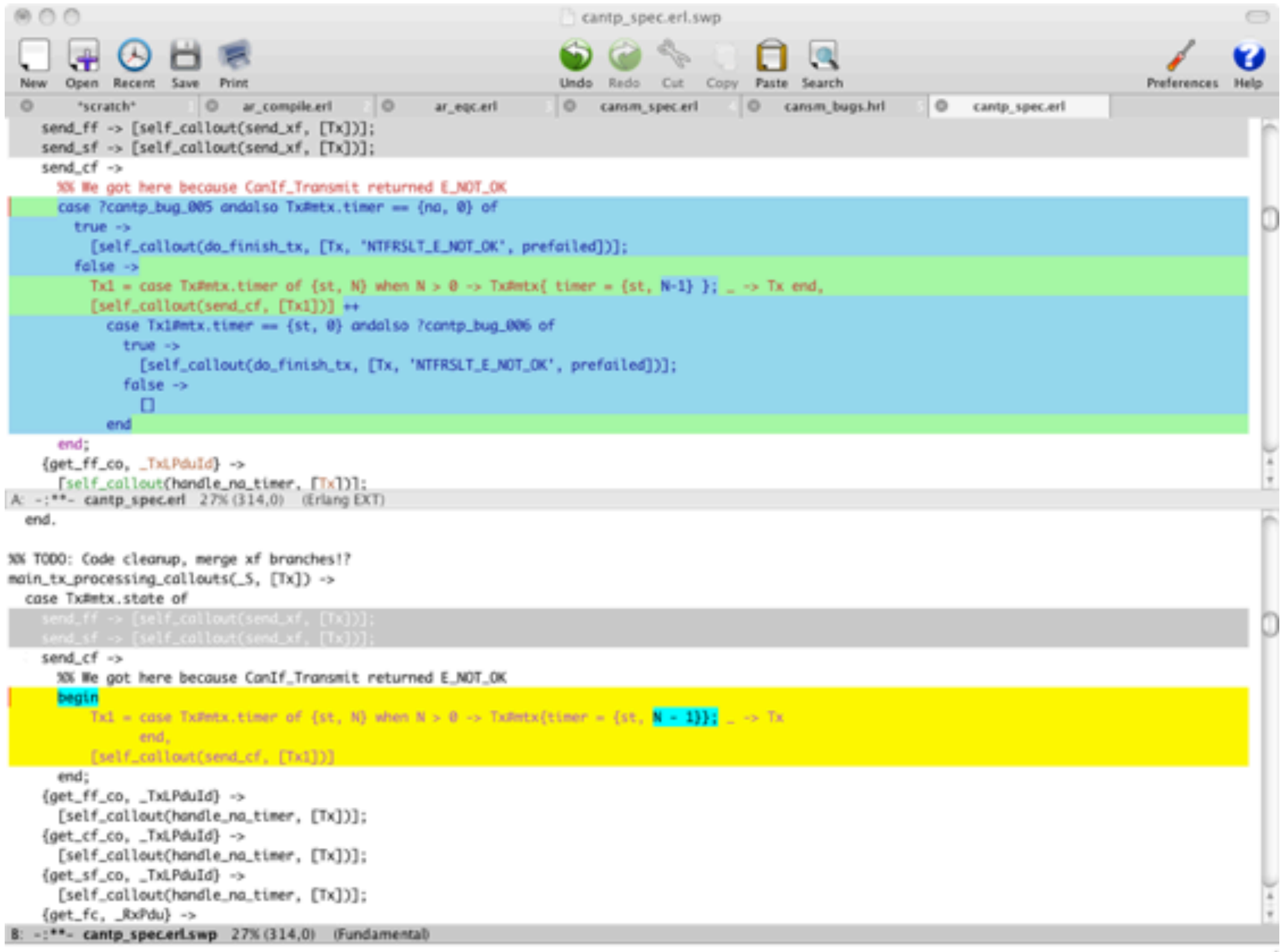
For buggy code, want to avoid hitting the same bugs all the time.

Add bug precondition macros …

… but want to remove in delivered code.

DSL + API.

And you can see the changes …



Pragmatic issues

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

Editor and IDE integration

Wider integration: comments, makefiles, tests, …

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

Technical

Meaning has been preserved.

Appearance has been preserved.

The appearance hasn't changed

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

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

```
{v1, v2, v3}

{v1,v2,v3}

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.

Learn layout for synthesised code from existing codebase.

Preserving meaning

What are we preserving?

Where are we preserving it?

Individual results or the refactoring tool itself?

Equivalences

Testing equivalence: ∀ test data [finite]

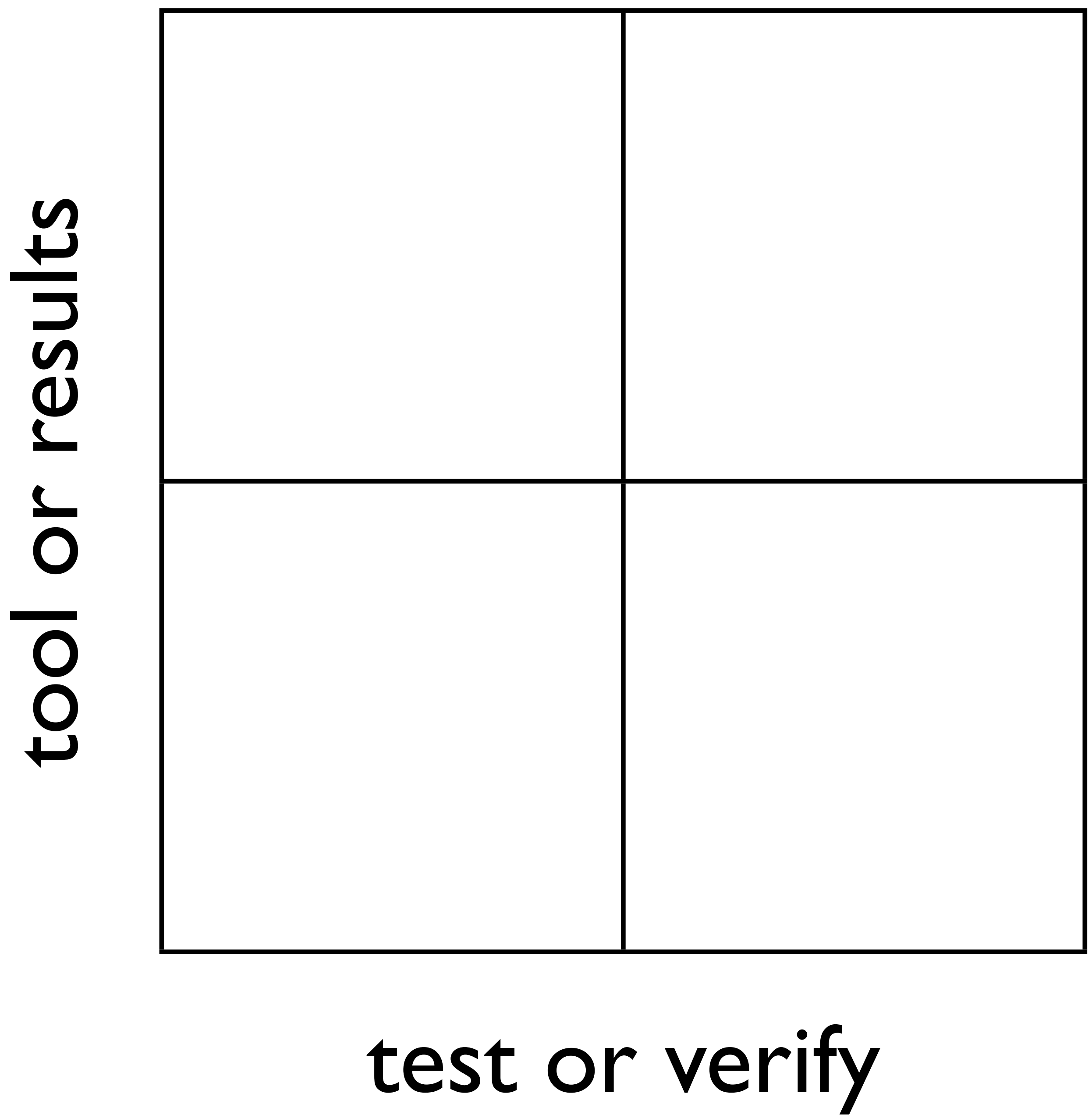
PBT equivalence: ∀ random test data [finite, but unbounded]

Extensional equivalence: ∀ input data [infinite]

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

Textual

Question: varieties of ↓: may be happy to converge on *more* inputs?

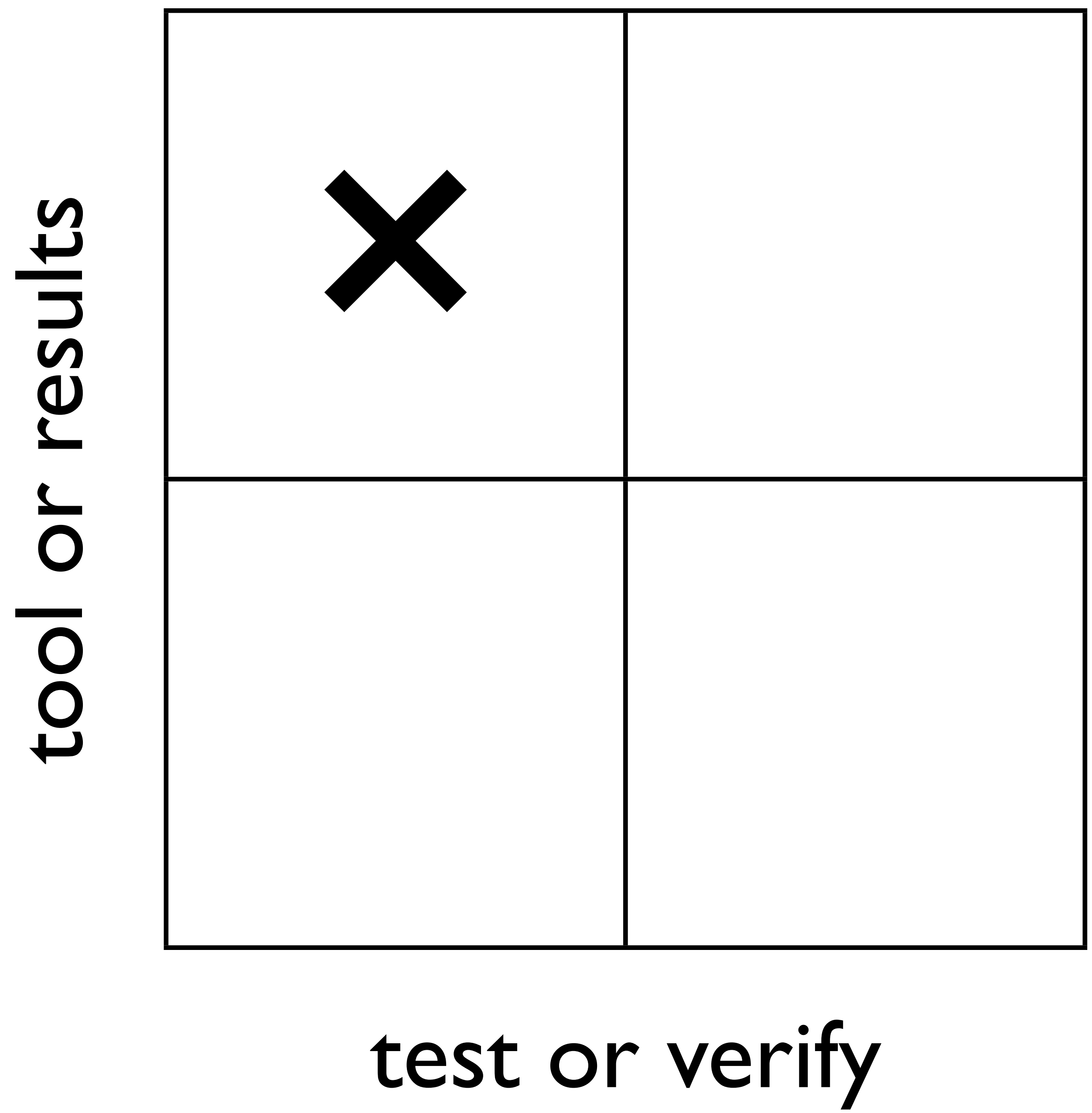

tool or results
test or verify

Testing

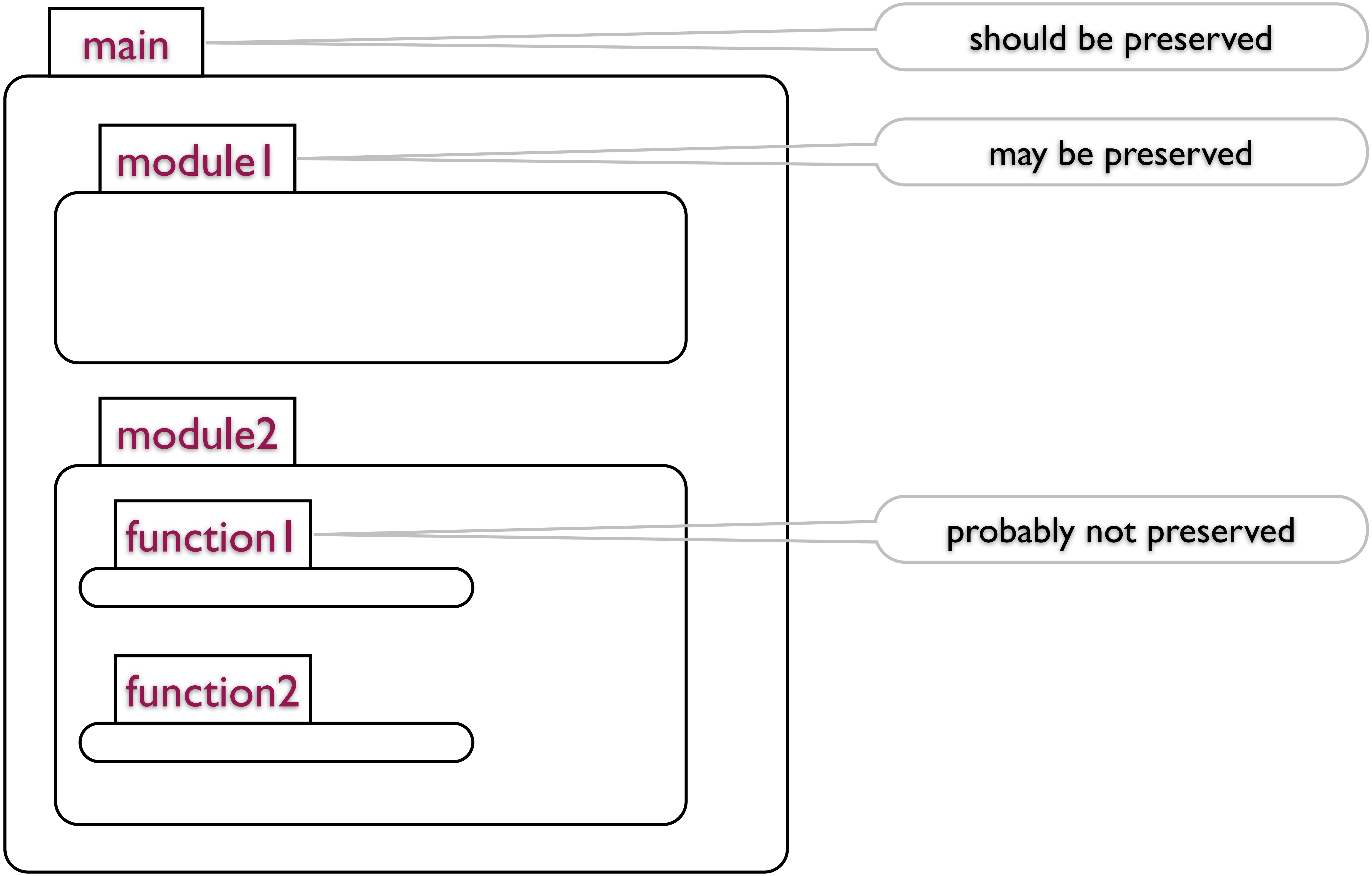
Testing the results of applying the tool …

Regression tests (and properties) for the system …

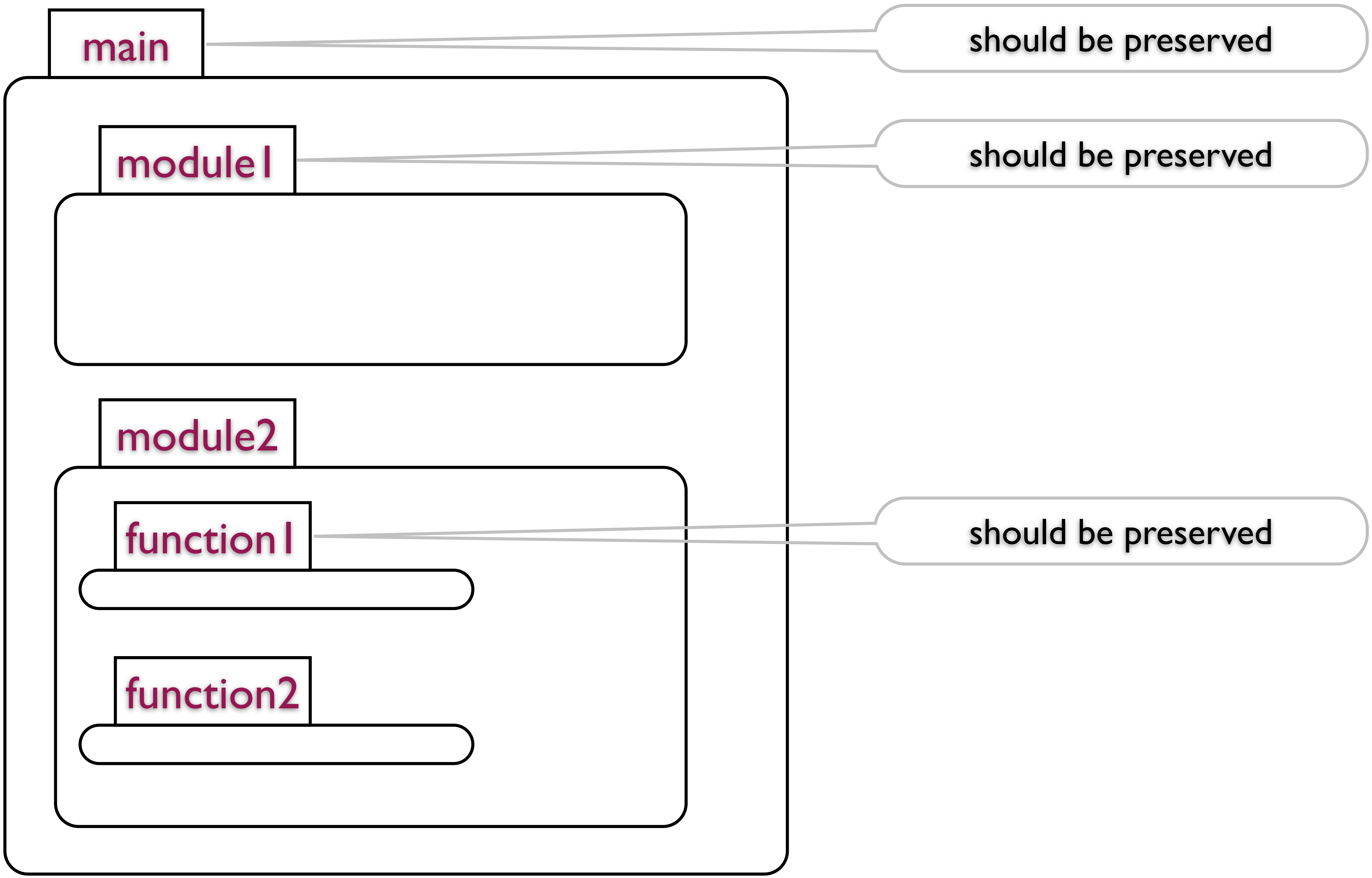
… and at module and function level (modulo refactoring).


tool or results
test or verify

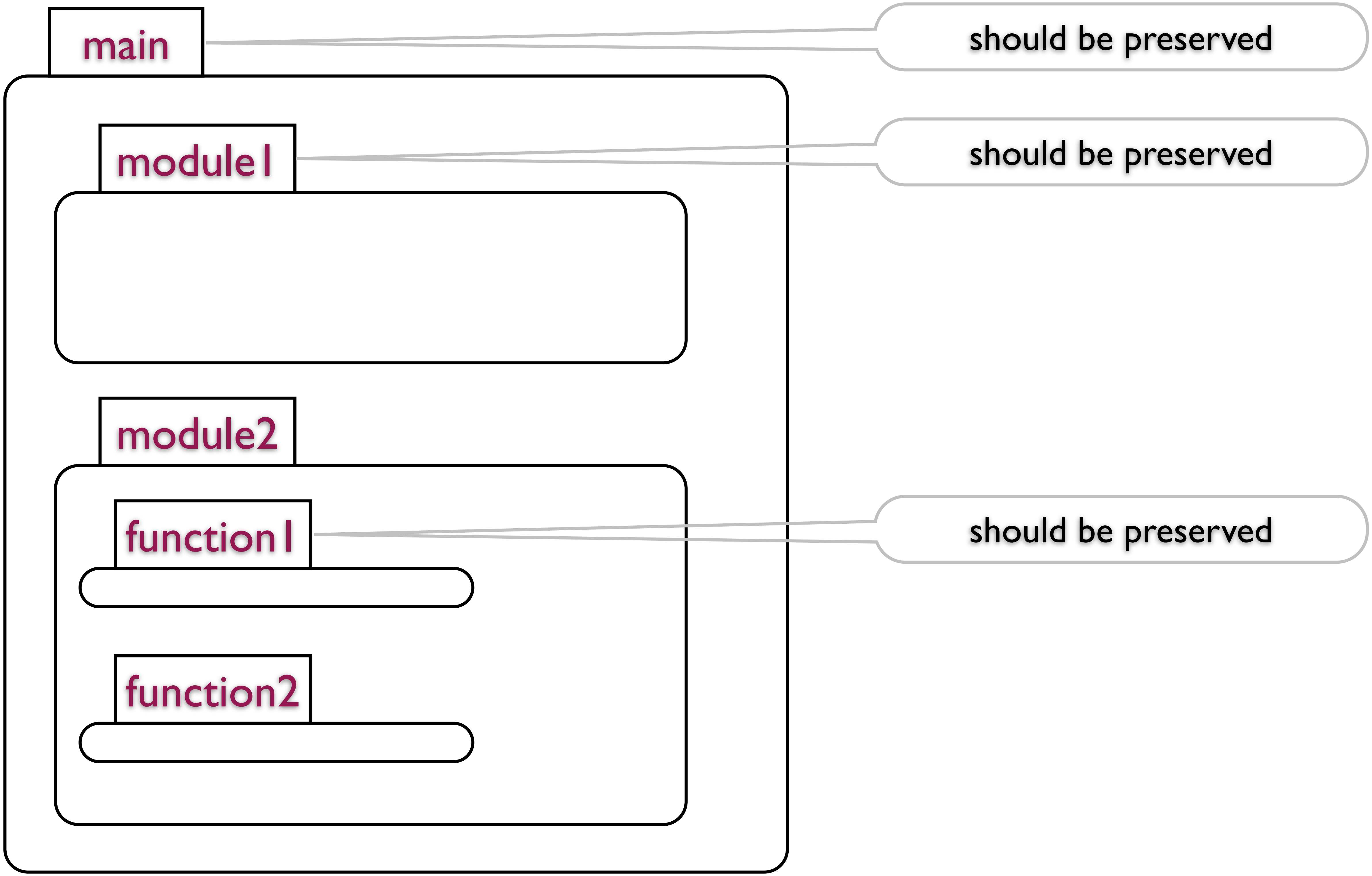
System, module and function



System + unit testing … refactor tests too



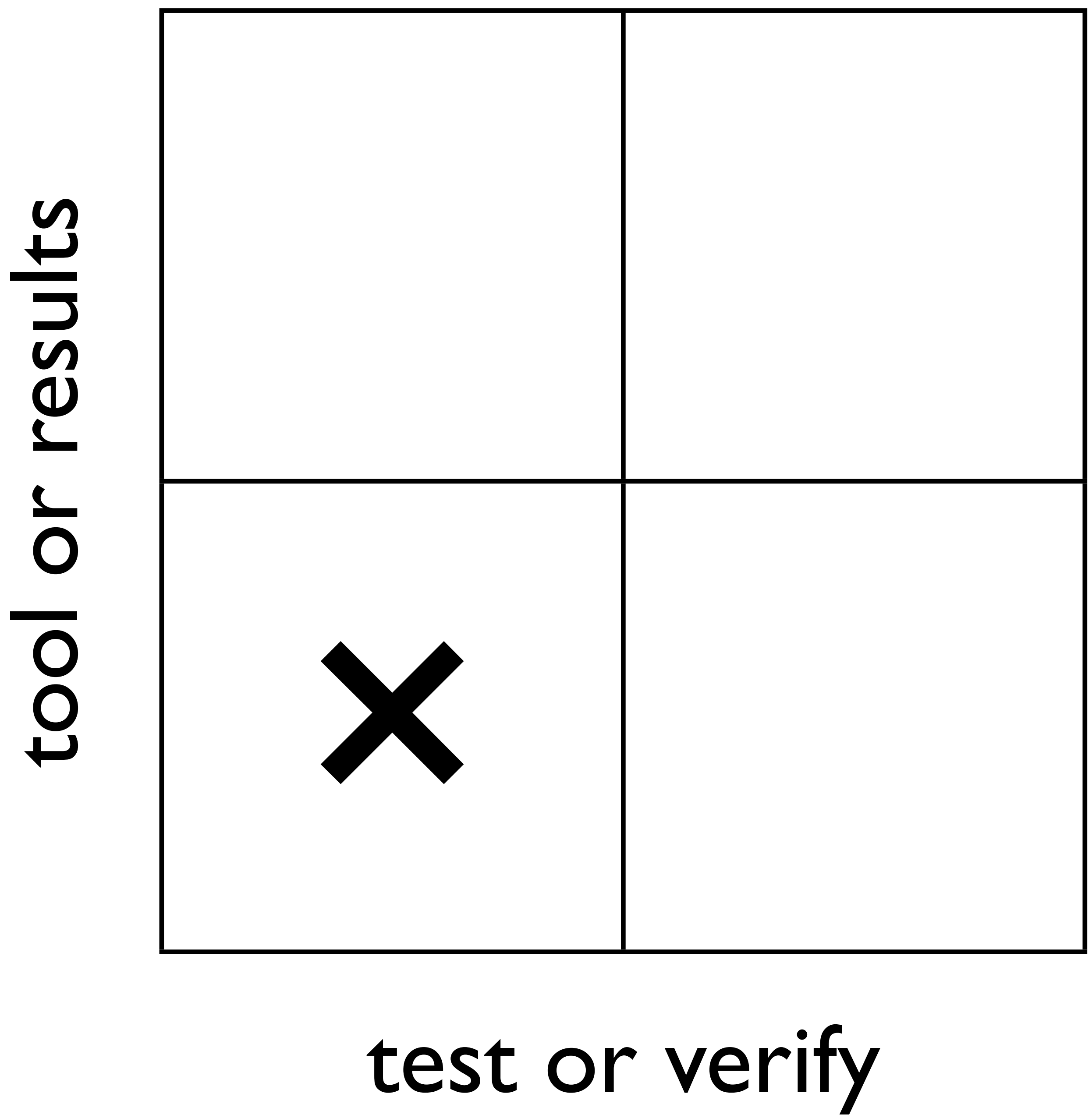
System + PBT … refactor properties too



… or testing the refactoring tool itself.

Generate programs as test data for the tool …

… together with refactorings and test data for the programs.

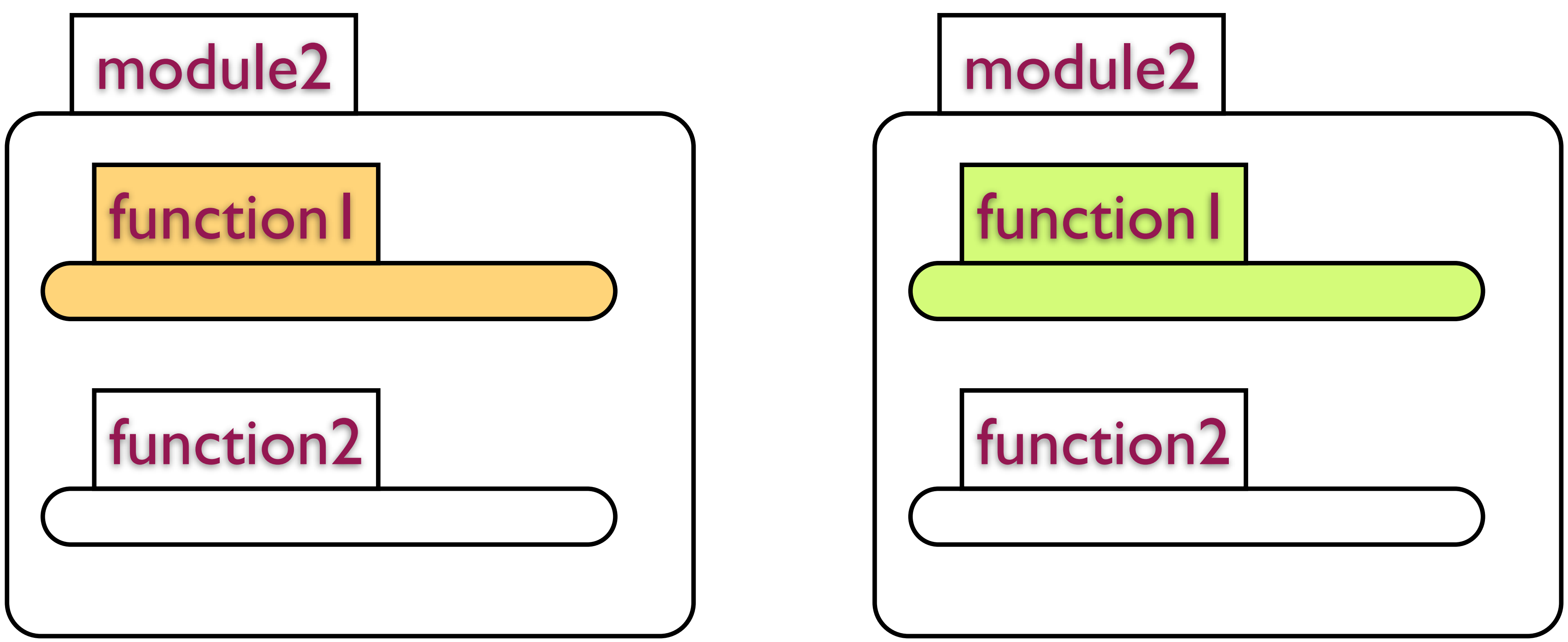

tool or results
test or verify

Testing two refactoring tools

Compare the results of tool1 and tool2 …

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

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

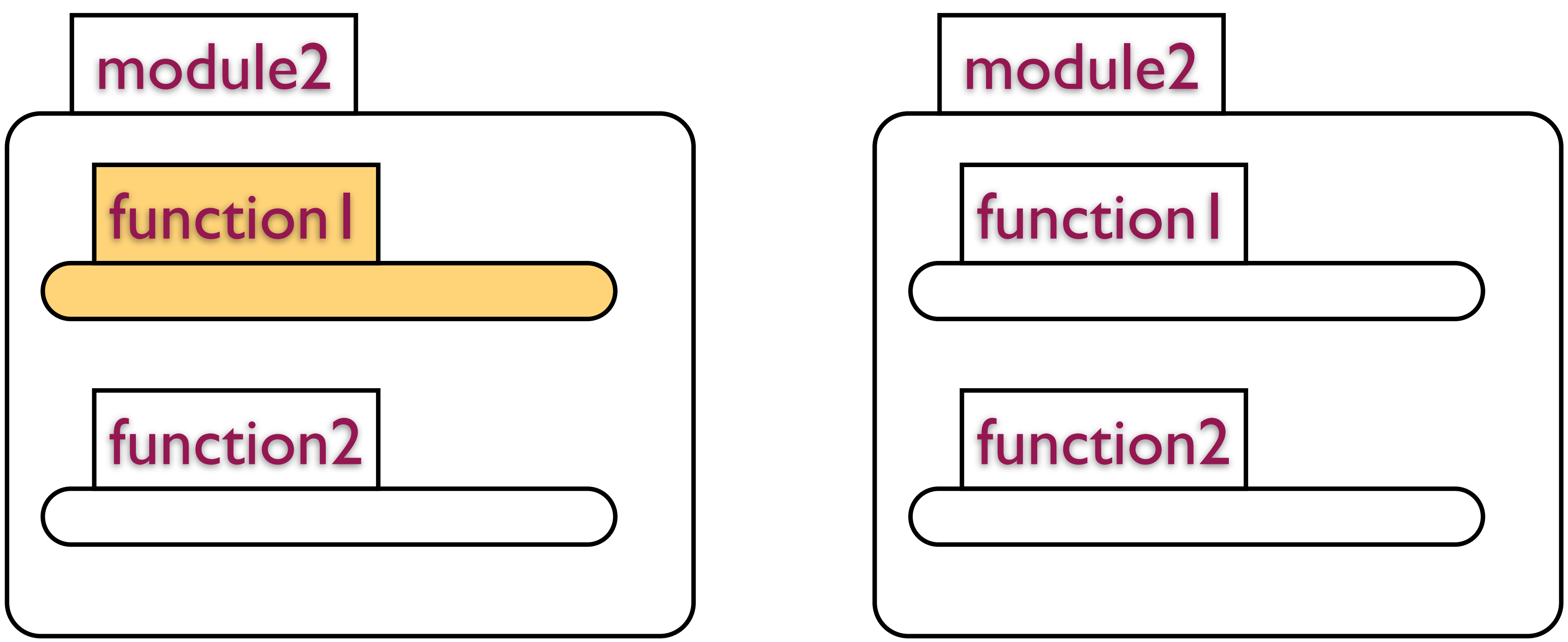


Testing one tool

Compare the results of function1 and function1 (unmodified) …

… using existing unit tests, or randomly-generated inputs

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

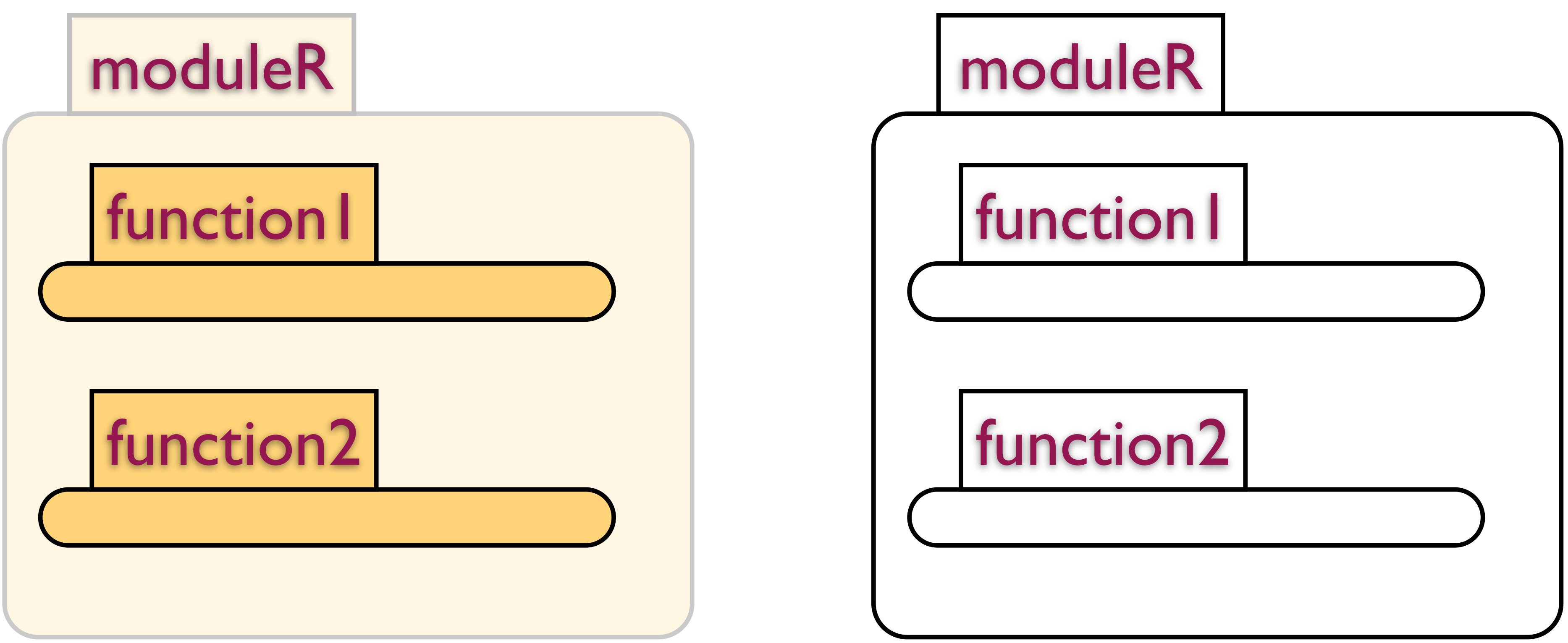


Fully random

Generate random modules,

… generate random refactoring commands,

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



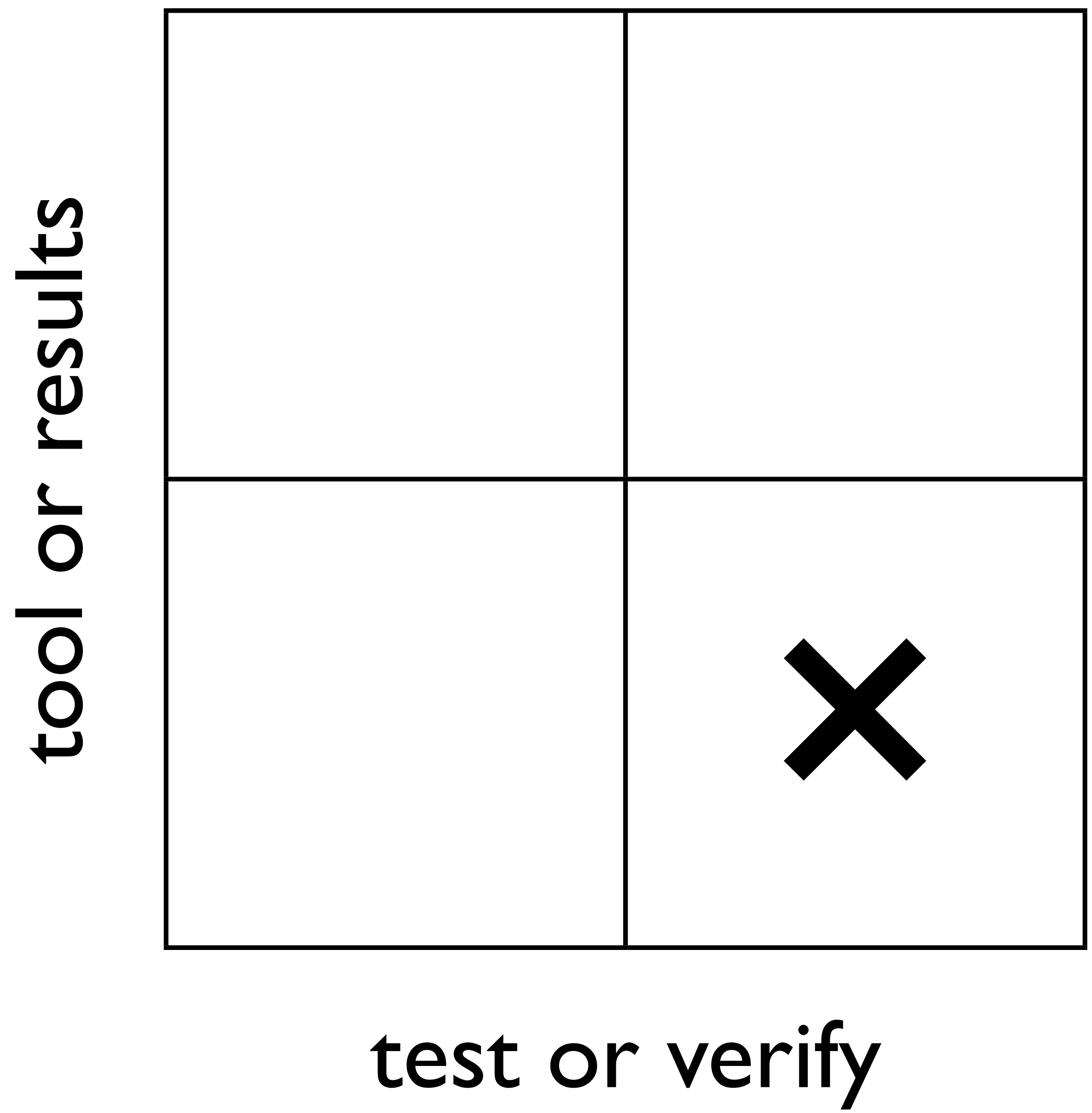
Verification

Verification

Tool-level verification for little languages …

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

Individual verifications: proof or counterexample.


tool or results
test or verify

Tool verification (with Nik Sultana)

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

Deep embeddings of small languages:

… potentially name-capturing λ-calculus

… PCF with unit and sum types.

Isabelle/HOL: LCF-style secure proof checking.

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

… principally to support pre-conditions.

Variable capturing substitution

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

Inductive definition of evaluation

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

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

$$x \notin FV(M)\ \frac{}{\lambda x.(M \cdot x) \simeq M}\ (\eta)$$

$$\frac{}{M \simeq M}\ (\textsc{Refl}) \qquad \frac{N \simeq M}{M \simeq N}\ (\textsc{Symm})$$

$$\frac{M \simeq M' \qquad M' \simeq N}{M \simeq N}\ (\textsc{Tran})$$

Extract a (local) definition

The composition of four steps

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

Extract a (local) definition … formally

$$
\begin{array}{l}
g \notin FV\,L \;\wedge \\
\neg Rec\,(g := N) \;\wedge \\
g \sharp (f := M) \;\wedge \\
N \subseteq_{\Lambda} M \;\wedge \\
\neg Captures\,fix\,f \;\wedge \\
\neg Captures\,L\,g \;\wedge \\
\neg Captures\,N\,f \;\wedge \\
\neg Captures\,L\,M \;\wedge \\
\neg Captures\,N\,M \;\wedge \\
\neg Captures\,(letrec\,f := (letrec\,g := N\ in\ M)\ in\ L)\,N \;\wedge \\
\neg Captures\,L\,(M[g:N]) \;\wedge\; \neg Captures\,N\,(M[g:N]) \;\longrightarrow
\end{array}
$$

$$letrec\,f := M\ in\ L \simeq letrec\,g := N\ in\ (letrec\,f := M[g:N]\ in\ L)$$

PCF + union: expand type example

If

- $\Gamma \triangleright N :: S \;\wedge\; \Gamma \triangleright x :: T \;\wedge\; \Gamma, y : T' \triangleright L :: T \;\wedge$
  $\Gamma \triangleright M :: T$
- $\neg \mathit{Captures}\, N \,\langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle \;\wedge$
  $\neg \mathit{Captures}\, N\, M \;\;\wedge\; \neg \mathit{Captures}\, L\, M$
- $x' \notin FV\, M \wedge y \notin FV\, M \wedge x \notin FV\, L$

Then

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

$\mathit{let}\; x : T + T' := \mathit{inL}_{T+T'}\, M\; \mathit{in}\; N[\langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle / x] \;:: S$

Full tool verification revisited

Tool-level verification for full scale tools?

This requires, at least:

- Meta-theory for a real language
- Semantics for a real language

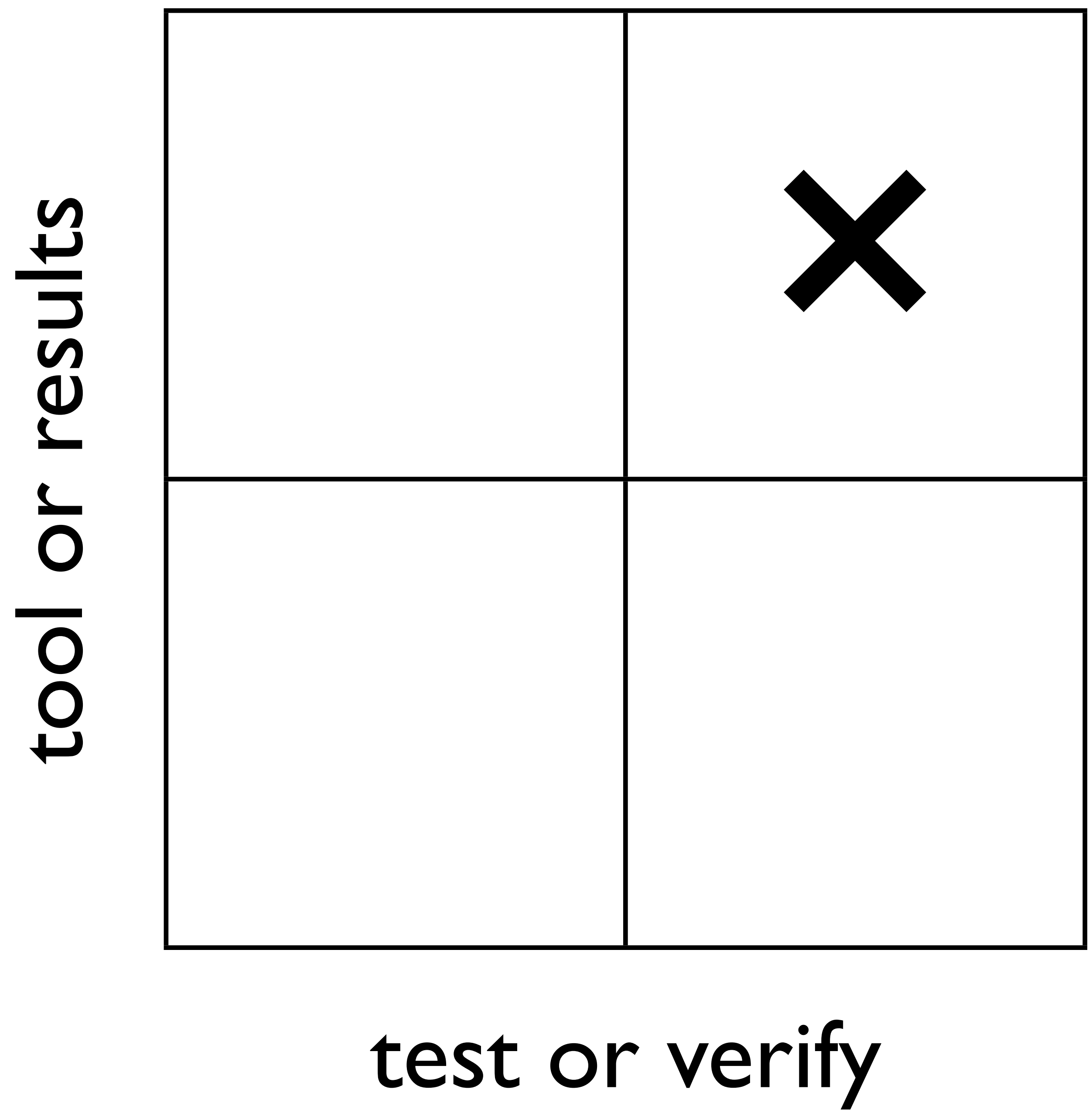
Idea (with Nik Sultana)

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

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

Again, will be a proof for a small language …

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


tool or results
test or verify

Automatically verify instances of refactorings

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

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

Manifestly clear what is being checked.

The approach delegates trust to the SMT solver …

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

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

Example

```
module Before where

h :: Integer->Integer->Integer

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

g :: Integer->Integer

g x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

Example: renaming

```haskell
module Before where

h :: Integer->Integer->Integer

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

g :: Integer->Integer

g x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

```haskell
module After where

h :: Integer->Integer->Integer

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

k :: Integer->Integer

k x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

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

module RefacProof where

import Data.SBV
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

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

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

g :: Integer->Integer

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

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

module RefacProof where

import Data.SBV
```

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

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

g :: Integer->Integer

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

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

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

k :: Integer->Integer

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

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

module RefacProof where

import Data.SBV
```

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

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

g :: Integer->Integer

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

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

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

k :: Integer->Integer

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

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

f = uninterpret "f"

-- Properties

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

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

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

g :: Integer->Integer

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

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

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

k :: Integer->Integer

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

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

f = uninterpret "f"

-- Properties

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

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

```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 = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

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

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

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

k :: Integer->Integer

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

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

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

g :: Integer->Integer

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

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

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

k :: Integer->Integer

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

```haskell
f = uninterpret "f"

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

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

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

g :: Integer->Integer

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

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

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

k :: Integer->Integer

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

```haskell
f = uninterpret "f"

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

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

Automatically verify instances of refactorings

Feasible … and open.

Compare with the task of general proofs, which requires …

… semantics and meta-theory for a real language

Can we extract evidence in the positive case, too?

Guaranteeing API and DSL?

API provides a general transformation framework …

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

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

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

DSL - correctness is ensured by correctness of component refactorings.

Is the approach functional or general?

Extended repertoire of expression-level refactorings.

These are local, and should be amenable to automated verification.

Structural refactorings similar for OO and other examples.

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

Thank you

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

ProTest property based testing

RELEASE