SIMON THOMPSON

LANGUAGE-INDEPENDENT REFACTORINGS THROUGH LANGUAGE-SPECIFIC REWRITES

Work with

Dániel Horpácsi, Judit Köszegi,
Péter Bereczky, Márton Sevella,
Dominik Katko and the HARP team

Collaboration with

Erlang Ecosystem Foundation, erlang_ls team, Rotor project, UKRI/EPSRC, Nik Sultana, Univ. of St Andrews

INTRODUCTION

Refactoring

Transforming code
to improve it in some way,
without changing what it does.

In practice

Transforming a large body of text, so that it is still recognisable & acceptable.

Under the hood

Working with a complex semantic object that includes types, bindings, effects, etc.

Requirement

We have to reconcile editing the complex semantic object with textual format being OK.

ASSURANCE

It's crucial that we get it right

We have to find ways of convincing users that our tools don't break their code.

Approaches

Is the code still OK?

Testing, SMT, proof, . . .

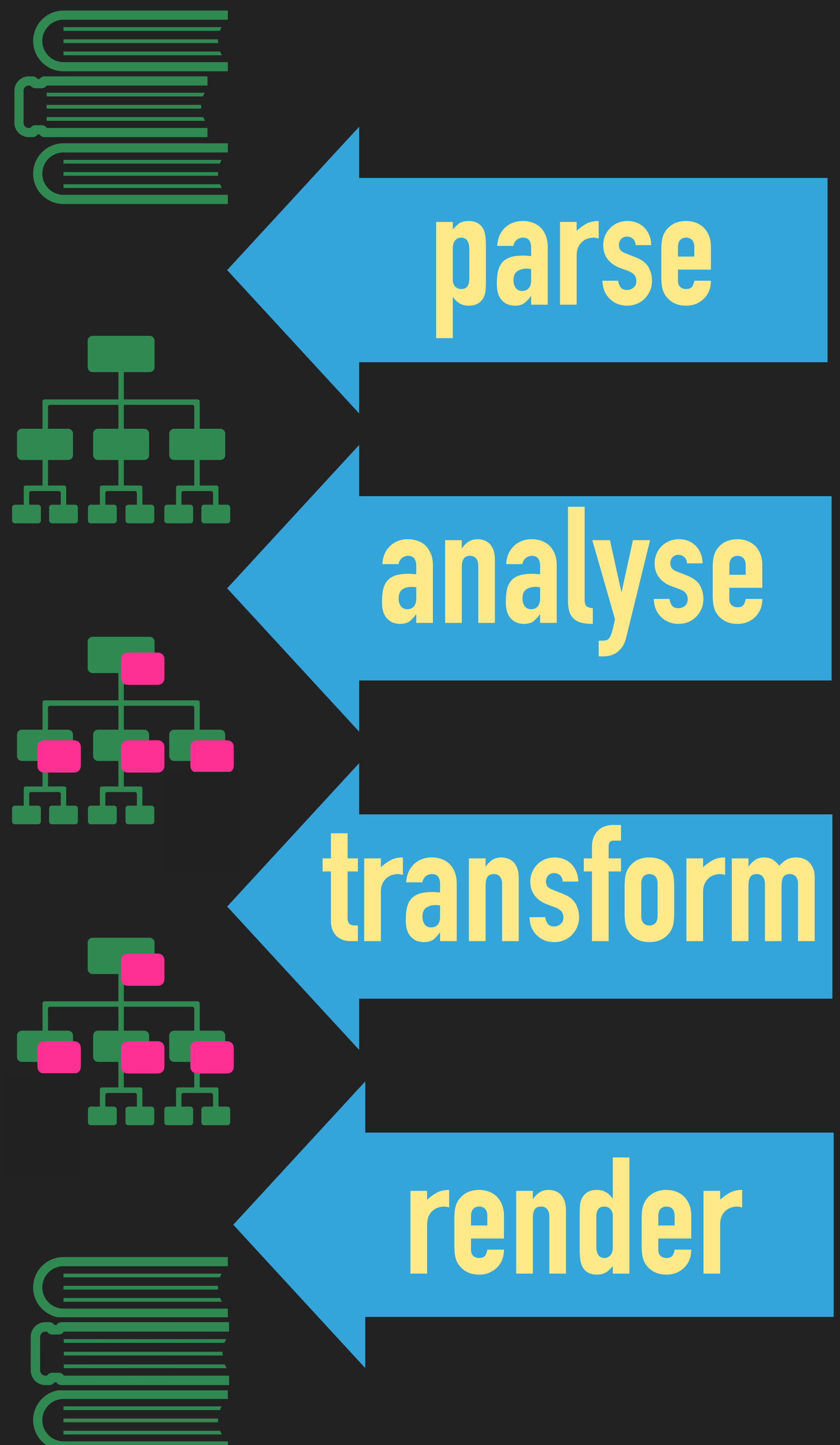
Is the system built right?

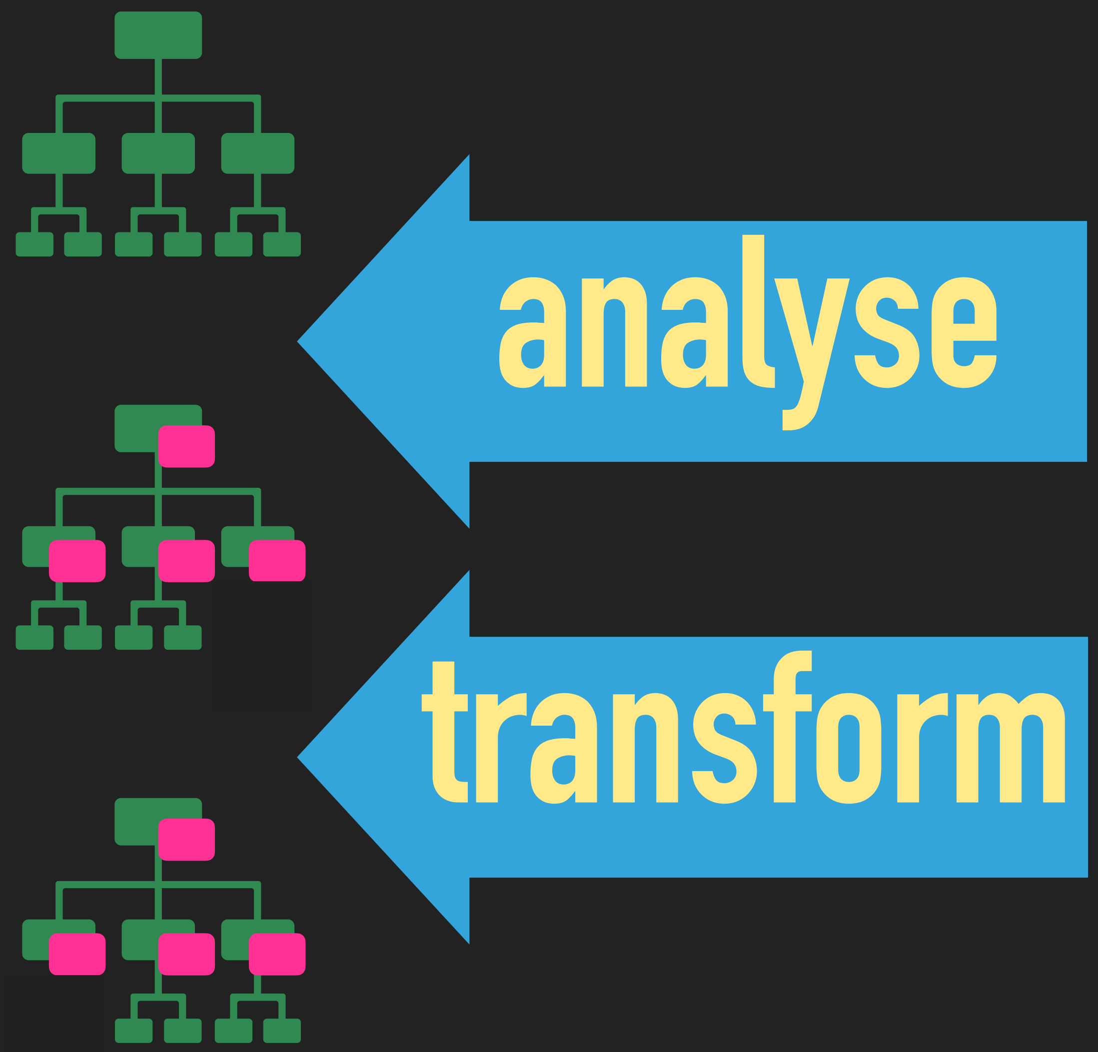
Engineering, proof

ARCHITECTURE

Building the tool right

Build tools that support ease of use & re-use, and are straightforward to implement.


parse
analyse
transform
render


analyse
transform

Compiler front-end

Static semantics

Types

Macros etc.

Architecture

Abstractions

Components

Libraries

Finding the right abstractions

Simple

General

Implementable

KEY INSIGHTS

Key insight #1

Language independent

Language dependent

Key insight #2

Layout independent

Layout dependent

Not just for functional languages

The examples here are from functional languages, but other abstractions OK for e.g. OO.

EXAMPLE

Example refactorings

Renaming,

Generalisation,

Argument reordering, . . .

Function transformation scheme

F(pat) = res

... F(args) ...

Function

F(pat) = res

. . . F(args) . . .

Transformation

Rename

Reorder args

Regroup args

Generalise

Language independent

Describe these examples in a language-independent way

Language independent

Describe these examples in a language-independent way only by hiding complexity . . .

Haskell

```haskell
f x y = x-y

g x = x `f` x

h x xs = map (f x) xs

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

Erlang

```erlang
f(X,Y) -> X-Y.

p(X) -> spawn(code,f,[2,X]).

g(X) -> X-1.

h(Xs) -> lists:map(fun g/1,Xs).
```

Complexity of application

Partial application, symbolic references, DIY infix, . . .

different in each language

THE FUNCTION TRANSFORMATION

Solution

Generic transformation

Language-specific rewrites

Function transformation scheme

$$F_{new}(pat) = res$$

$$F_{old} = \ldots F_{new} \ldots$$

Function

f(X) -> X+3

... f(A) ...

Generalise

f(X,Y) -> X+Y

... f(A,3) ...

Function transformation scheme

$f_{new}(X,Y)$ -> X+Y

f_{old} = fun (X) -> $f_{new}(X,3)$ end

Applying the scheme

$f_{new}(X,Y)$ -> X+Y

f_{old} = fun (X) -> $f_{new}(X,3)$ end

$g_{old}(Z)$ -> f(Z+2)

Applying the scheme

$f_{new}(X,Y) \rightarrow X+Y$

$f_{old} = \text{fun } (X) \rightarrow f_{new}(X,3) \text{ end}$

$g_{old}(Z) \rightarrow f(Z+2)$

Applying the scheme

f_{new}(X,Y) -> X+Y

f_{old} = fun (X) -> f_{new}(X,3) end

g_{old}(Z) -> f(Z+2)

g_{new}(Z) -> (fun (X) -> f_{new}(X,3) end)(Z+2)

Applying the scheme

f_{new}(X,Y) -> X+Y

f_{old} = fun (X) -> f_{new}(X,3) end

g_{old}(Z) -> f(Z+2)

g_{new}(Z) -> (fun (X) -> f_{new}(X,3) end)(Z+2)

Applying the scheme

$f_{new}(X,Y) \text{ -> } X+Y$

$f_{old} = \text{fun } (X) \text{ -> } f_{new}(X,3) \text{ end}$

$g_{old}(Z) \text{ -> } f(Z+2)$

$g_{new}(Z) \text{ -> } (\text{fun } (X) \text{ -> } f_{new}(X,3) \text{ end})(Z+2)$

$g_{new}(Z) \text{ -> } f_{new}(Z+2,3)$

Correctness proof obligation

$f_{old}(X) \text{ -> } X+3$

$f_{new}(X,Y) = X+Y$

$f_{old} \stackrel{?}{=} \text{fun } (X) \text{ -> } f_{new}(X,3) \text{ end}$

Transformation	Rewrites
One transform per refactoring	Many rewrites to tidy up the code …
One proof per refactoring	… but only need to be proved once.

Formalisation

Proof assistant: Coq

Formalise language: CoreErlang

Formalise framework: AML

API MIGRATION

Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs

Huiqing Li
School of Computing
University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
School of Computing
University of Kent, UK
S.J.Thompson@kent.ac.uk

ABSTRACT

Wrangler is a refactoring and code inspection tool for Erlang programs. Apart from providing a set of built-in refactorings and code inspection functionalities, Wrangler allows users to define refactorings, code inspections, and general program transformations for themselves to suit their particular needs. These are defined using a template- and rule-based program transformation and analysis framework built into Wrangler.

This paper reports an extension to Wrangler's extension framework, supporting the automatic generation of API migration refactorings from a user-defined adapter module.

Categories and Subject Descriptors

D.2.3 [**SOFTWARE ENGINEERING**]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement

General Terms

Languages, Design

Keywords

Erlang, refactoring, API migration, Wrangler, software engineering, template, rewrite rule.

1. INTRODUCTION

Most software will evolve, and this will often change the API of a library, and such changes could potentially affect all client applications of the library, both locally and remotely. API migration is a process of refactoring, but API migrations are not generally supported by refactoring tools due to the specifics of each particular migration, and so the transformations required tend to be done manually by the maintainers of the client code, risking incorrectness.

This paper presents our approach to automating the implementation of API migration for Erlang. This work is built on top of Wrangler, a refactoring and code inspection tool for Erlang programs, but we note that the approach applies to other languages equally well. One of the features that distinguishes Wrangler from other refactoring tools is its user-extensibility, given by a template- and rule-based program analysis/transformation framework, allowing users to express their intentions using Erlang concrete syntax.

Our approach to automatic API migration works in this way: when an API function's interface is changed, the author of this API function implements an *adapter function*, defining calls to the old API in terms of the new. From this definition we automatically generate the refactoring that transforms the client code to use the new API. This refactoring can be supplied by the API writer to clients on library upgrade, allowing users to upgrade their code automatically.

As a design principle, we try to limit the scope of changes as much as possible, so that only the places where the 'old' API function is called are affected, and the remaining part of the code is unaffected. One could argue that the migration can be done by *unfolding* the function applications of the old API function using the adaptor function once it is defined. However, the code produced by this approach would be a far cry from what a user would have written. Instead, we aim to produce code that meets users' expectations.

The paper is organised thus: Sec. 2 introduces a running example, and Sec. 3 gives a brief overview of Wrangler and its template- and rule-based framework. Automated API migration in Wrangler is reported in Sec. 4, related work is covered in Sec. 5, and the paper is concluded in Sec. 6.

2. EXAMPLE: REGULAR EXPRESSIONS

As a running example we take the implementation of *regular expressions* in Erlang; the `regexp` library has been deprecated, and users are expected to use the `re` library, which has a somewhat different application programmer interface.

For instance, the function `match` from the `regexp` library is used to find the first longest match of regular expression `RegExp` in a `String`. If the match succeeds, the function returns a tuple `{match, Start, Length}` where `Start` is the starting position of the match, and `Length` is the length of the matching string; if the match fails it returns `nomatch`. Fig. 1 shows two examples that use the function; note that it would be possible to rewrite the `case` expressions in various different ways without changing their meaning.

Replacing uses of `match` in Fig. 1 with the corresponding functions in the `re` library gives Fig. 2. In particular, the replacement for `match` would be the `run` function with the option `global` set. The function `run` is different from `match` not only in the name, but also in inputs and outputs. The

API migration

Change in library API.

Erlang example: from `regexp` to `re`.

How to refactor client code to accommodate this?

CONCLUSION

Solution

Generic transformation

Language-specific rewrites

EXTRA MATERIAL

OTOH: human factors

90% correct better than nothing

Layout change unacceptable

I trust what X does

OTOH: programming language

If it type checks then it's OK

If it runs, then fine

Function	Rename
F(pat) = res	G(pat) = res
. . . F(args) . . .	. . . G(args) . . .

Function	Swap
F(X,Y) = res	F(Y,X) = res
… F(A,B) …	… F(B,A) …

Function

Generalise

$F(X) = X+3$

$F(X,Y) = X+Y$

... F(A) ...

... F(A,3) ...

Applying the scheme

f_{new}(X,Y) -> X+Y

f_{old} = fun (X) -> f_{new}(X,3) end

g_{old}(Zs) -> map(fun f/1,Zs)

Applying the scheme

f_{new}(X,Y) -> X+Y

f_{old} = fun (X) -> f_{new}(X,3) end

g_{old}(Zs) -> map(fun f/1,Zs)

Applying the scheme

f_{new}(X,Y) -> X+Y

f_{old} = fun (X) -> f_{new}(X,3) end

g_{old}(Zs) -> map(fun f/1,Zs)

g_{new}(Zs) -> map(fun (X) -> f_{new}(X,3) end,Zs)

Applying the scheme

f_{new}(X,Y) -> X+Y

f_{old} = fun (X) -> f_{new}(X,3) end

g_{old}(Zs) -> map(fun f/1,Zs)

g_{new}(Zs) -> map(fun (X) -> f_{new}(X,3) end,Zs)