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

Funding *Simon Thompson*: With support from the UK Engineering and Physical Sciences Research Council awards EP/N028759/1, EP/C524969/1, GR/R75052/01 and EP/T014512/1.
Dániel Horpácsi: With support from the NRDI Fund of Hungary and financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

1 Introduction

Our subject here is not new: Eelco Visser initiated a discussion of language-independence transformations [25] in the millennium year; it is a pleasure and an honour to join the conversation that he began.

Refactoring tools are a particularly sensitive part of a programmer's toolkit, since they can make large-scale modifications to code, and yet are expected not to change the observable behaviour of the system. Users therefore need to be given assurance about the safety of using a tool. Assurance generally comes in two complementary forms, through verification and through architecture.

Arguments can be made about the *correctness* of the transformations made. These can be black box, checking the original against the refactored code without examining how the transformation is performed. At minimum this is achieved by regression testing, but can be augmented by checking equivalence more generally [8, 9]. Looking inside the implementation it is also possible – at least in principle – to prove the correctness of that transformation [23].

[1] Corresponding author

© Simon Thompson and Dániel Horpácsi;
licensed under Creative Commons License CC-BY 4.0

These checks will apply to particular *instances* of a refactoring in the case of regression testing, while verification should establish the correctness of the transformation in itself, that is for *all* possible instances. Testing can also be used in the latter case, e.g. by generating random transformations of an arbitrary system, and testing the "old" code against the "new" with random inputs [5].

An alternative source of assurance is in the *architecture* of the refactoring tool itself. At its heart, any refactoring tool works by transforming a complex data structure (AST or database) representing a program into a related structure. With no further thought, each refactoring can be constructed anew, by means of an *ad hoc* recursive function. But this can be improved. Firstly, following Eelco Visser's work [25], it is possible to take a higher-level view of the way the data structure is traversed, using a *strategic* approach in something like Stratego [4], Strafunski [15] and related systems.

Secondly, we can see a commonality between the refactorings themselves, and this is the approach we outline here. With this perspective, a class of refactorings can all be performed with a single implementation. This, in turn, reduces the burden on users wishing to assure themselves of the soundness of the implementation, and indeed has positive consequences for formal verification of the refactoring transformations too.

In the remainder, we first introduce the separation of generic and specific elements of refactoring definitions in §2. Then in §3 we argue that the generic parts in many cases can be made language-independent. Finally, §5 discusses some related work and §6 concludes.

2 Refactoring = Substitution + Rewriting

A class of refactorings in Erlang are concerned with the definition and subsequent use of functions, including, among others, renaming and generalisation. In this section we show how these can be described in a common way, and the implications of this for verification.

2.1 Function renaming

Consider the example of function renaming.

Before renaming

```
f(X) -> X+1.
g(Y) -> f(Y+2) - f(Y-2).
```

After renaming

```
h(X) -> X+1.
g(Y) -> h(Y+2) - h(Y-2).
```

The transformation is described by showing how the function *definition* is changed, and also explaining how to change each *use* of the function. We do that thus:

Define the modified function:

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

Implement the old function using the new:

```
f = fun(X) -> h(X) end
```

How does this describe the refactoring? We can use the implementation of the old function in terms of the new to give us the new code, in a series of steps, thus:

```
g(Y) -> f(Y+2) - f(Y-2).
-- by substitution giving
g(Y) -> (fun(X) -> h(X) end)(Y+2) - (fun(X) -> h(X) end)(Y-2).
-- and by rewriting (beta-reduction, here)
g(Y) -> h(Y+2) - h(Y-2).
```

Rewriting stops at this point: we don't want, or need, to inline `h`, since we want to be faithful to the original program that contains a function call to `f` here.

It is important to note that this approach works for other uses of the function `f`, including as `fun` arguments to higher-order functions, and, with some preliminary eta-expansion [2], in calls to `spawn` the function.

2.2 Function generalisation

Now we look at a second example, function generalisation, and see that it fits the same pattern.

Before generalisation

```
f(X) -> X+3.
g(Xs) -> lists:map(fun f/1,Xs).
```

After generalisation

```
f(X,Y) -> X+Y.
g(Xs) ->
  lists:map(fun(X)-> f(X,3) end, Ys).
```

This transformation is described by showing how the function *definition* is changed, and also explaining how to change each *use* of the function. We do that thus:

Define the modified function:

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

Implement the old function using the new:

```
f = fun(X)-> f(X,3) end
```

How does this describe the refactoring? We can use the implementation to give us the new code, in a series of steps, thus:

```
g(Xs) -> lists:map(fun f/1,Xs).
-- by substitution giving
g(Xs) -> lists:map(fun(X)-> f(X,3) end, Ys).
-- after which no rewriting is necessary
```

The implementation of the old function in terms of the new is denoted by `=` rather than `->` to emphasise that this is a semantic equivalence rather than program code defining a function, since the LHS refers to the "old" version of the code and the RHS to the "new".[3]

As earlier, this approach will handle "regular" function applications, in which the `fun` expression will be removed by rewriting, as well as calls to `apply` and `spawn`.

2.3 Other function-oriented examples

Other examples include function unfolding, reordering and regrouping of arguments, adding or removing an argument. We leave it to the reader to verify this. In each case, the general pattern is to present the **new definition** and to describe how to **implement the old function using the new.**

We examine other kinds of refactoring in Section 4.3.

[2] Transforming `fun f/1` into `fun(X)->f(X) end`.

[3] It is a peculiarity of Erlang that the two versions of `f` can co-exist, as functions with the same name but different arity are considered to be distinct.

2.4 Verification

What do we need to establish for the transformed code to be equivalent to the original? The verification factors into two parts:

- For **each specific refactoring** it is necessary to ensure that the "old" function is implemented correctly in terms of the "new". Specifically, the replacement becomes a *proof obligation*. In the case of renaming, we require that `f` has the same behaviour as
  ```
  fun(X) -> h(X) end
  ```
  when `h` is defined thus:
  ```
  h(X) -> X+1.
  ```
  Similarly for other refactorings.
- On the other hand, **every refactoring** also depends on the correctness of the rewriting rules, such as beta-reduction, eta-expansion, removal of syntactic sugar, etc., which are applied to "tidy up" the resulting code in each case.

3 Towards generic, language-independent refactorings

Refactoring has a very different character in different programming languages; to take one example, [12] compares refactoring in two functional programming languages: Haskell and Erlang. Because of this, the first author was always sceptical about a *language-independent* approach to refactoring. In this Section we argue that our approach of substitution and rewrite allows us to split refactorings into language-independent and language-dependent parts.

At the **language independent** level is a concept like *function application*; function applications can be transformed by *defining the old function in terms of the new*, as described earlier. On the other hand, the **particular** form of function application in different languages varies widely, for example.

- In **Haskell** function applications can be infix `x 'f' y` as well as prefix `f x y`, and functions – prefix or infix– can also be partially applied, as in the expressions `map (f x) xs` and `map (x 'f') xs`.
- While **Erlang** does not contain infix function or partial applications, functions are passed as arguments using the "function/arity' idiom `fun/N`, but also be can be referenced by *atoms* in some special functions, such as `spawn`.

These differences can be dealt with by means of rewriting, as we saw with Erlang earlier. Consider the Haskell example

```
f x y = x+y

g z xs = map (f z) xs
```

where the order of arguments to `f` is reversed, so that the original `f` is implemented in terms of the new thus `\x y -> f y x`. Applying this transformation to the definition of `g` gives

```
g z xs = map (f z) xs
-- substituting the definition of f
g z xs = map ((\x y -> f y x) z) xs
-- by beta reduction
g z xs = map (\y -> f y z) xs
```

This leaves a lambda ('\') in the refactored expression, but this is unavoidable. While this example might have been handled better using the `flip` function, this approach generalises to any permutation (or tupling) of the arguments by introducing the appropriate, unnamed, equivalent of `flip` as the lambda expression.

If, on the other hand, we had renamed `f` to `h`, the redefinition of `f` would be `\x y -> h x y`, and the refactoring would completely eliminate the introduced lambda thus:

```
g z xs = map (f z) xs
-- substituting the definition of f
g z xs = map ((\x y -> h x y) z) xs
-- by beta reduction
g z xs = map (\y -> h z y) xs
-- by eta reduction
g z xs = map (h z) xs
```

It is also possible to accommodate infix operations into this framework too. One option is to recognise `'f'` as a function syntax; alternatively, and preferably, we can *pre-process* the code prior to substitution. In this case we proceed thus:

```
g z xs = map (z 'f') xs
-- replacing the infix "syntactic sugar"
g z xs = map (infix f z) xs
-- substituting the definition of f
g z xs = map (infix (\x y -> h x y) z) xs
-- by eta reduction
g z xs = map (infix (\x -> h x) z) xs
-- by eta reduction
g z xs = map (infix h z) xs
-- reintroducing the infix "syntactic sugar"
g z xs = map (z `h`) xs
```

Language dependence can extend beyond syntactic sugar. For example, in Ocaml function names can appear in signatures and as arguments to functors, where parameters are identified by name rather than position. This impacts the way in which the scope of a renaming refactoring is identified, as explained in [18].

4 Discussion

The approach discussed here is based on some assumptions, and so has some advantages, as well as some limitations. We discuss these in more detail now.

4.1 Local *vs* global

Many refactorings are *local*, in the sense of being applied at a single point, such as a replacement of a double list traversal `map f. map g` by a single one `map (f.g)`, but it is *global* refactorings, whose effect might span multiple sites within multiple modules, that are more problematic to implement and to review, e.g. in a pull request.

We have concentrated on global refactorings here for that reason, but a rewriting approach plainly works well for implementing local refactorings too, as shown by the retrie [17] tool for Haskell. On the other hand, it is difficult to see most local refactorings as anything other than language specific.

4.2 Recursion *vs* iteration

How might the replacement of recursion by iteration be seen in this framework? To encapsulate recursion in general would require some kind of template language, but then an arbitrary recursion cannot be replaced by iteration. Once the recursion has a stylised form, this can be encapsulated in a combinator, as in

```
diffs xs = foldr (-) 0 xs
```

and then a transition to an iterative form can be given thus

```
diffs xs = foldl (flip (-)) 0 (reverse xs)
```

Further transformation can render the list reverse in an iterative way too. This has been expressed in the syntax of Haskell, but all functional languages contain cognates of lists and folding operations, and so, arguably, it has a language-independent aspect.

4.3 Other kinds of refactoring

A similar approach can be taken to *constructor-based* refactorings in languages like Haskell and OCaml. A constructor is like a function, except that it can be used on the "left hand side' of definitions in pattern matches, and this requires some limited form of rewriting on patterns to implement.

There are limits to the approach described here. For example, 'folding' function definitions, i.e. replacing instances of a function body by a call to the function necessitate replacing (an instance of) a complex expression, rather than a single term. In the short term, we aim more clearly to articulate the scope and limits of the approach.

5 Related work

5.1 Language-independence and genericity

The questions of language-independence and genericity for refactorings have been addressed before. Indeed, Eelco Visser initiated a discussion of this in the millennium year in *Language Independent Traversals for Program Transformation* [25], which described how strategic, traversal-based programming could achieve transformations such as change in bound variable names across functional and OO languages that could be subsets of Haskell and C++ respectively. Shortly after this Lämmel's *Towards Generic Refactoring* [11] took this explicitly to the example of function/abstraction extraction with an approach that performs a conceptual analysis of the categories of transformations and pre-conditions that are necessary for a generic treatment of a refactoring.

This approach is flexible and comprehensive, but it lacks completeness. While it can encompass the generic features that occur in multiple languages, and indeed adapt e.g. to the transition between expression- and statement-oriented languages, it does not support the particularities of different languages, such as operator sections in Haskell or the use of atoms to denote functions in Erlang, that our approach can handle.

While we have argued that our approach supports a degree of language independence, we would not claim that it directly supports multi-language refactoring [21], since that requires not only awareness of the separate semantics of a number of languages, but also their semantic interactions.

5.2 Verification of refactorings

A powerful approach to ensuring the correctness of refactorings is to ensure that they meet the set of constraints that embody (aspects of) the semantics of the programming language being refactored. This insight was first presented by Tip and colleagues in the context of preserving *type constraints* [24], and elaborated for Java by de Moor and Schaefer [20]. Steimann [22] presents a general theory of constraint-based refactoring, and outlines a program in which correct-by-construction tools can be built on top of constraint-based presentations of programming language semantics.

Pioneering work by Kniesel and Koch [10] examines the way that correct refactorings can be built by composing simpler parts that themselves preserve behaviour, or can be verified separately. Our approach is related, but differs in that different instances of the same refactoring will involve different rewrites, depending on the context of the instance: the composition is thus, in a sense, dynamic.

5.3 API migration

When an API is upgraded it can be taken for granted that the new API should afford all the functionality provided by the old version; this can be made concrete in an *adapter module* that defines the old in terms of the new.

While adaptation is enough to ensure that the client system continues to work, it has disadvantages. If an API evolves continually, then a series of adapter modules will stack up, and even in the case of a single adaptation, the code will be neither idiomatic nor natural. One approach to this is to generate transformations from the replacement code [16], which ensures that the explicit wrapper code disappears. This mechanism is extended by our approach, outlined in [13], where the replacement code is subsequently simplified by rewriting, e.g. removing case expressions when they can be resolved, or exception-handling code when that is unnecessary.

This adaptation can be complex, however, particularly in the case of object-oriented programming, and especially when the migration is from one API to another, unrelated one. Lämmel and colleagues outline this in a case study [1] of evolving a system, while providing a broad overview of previous approaches, as well as in this general exploration of two XML case studies here [2].

It turns out that the approach we outline in this paper can be seen as a particular case of the work presented in [13], viewing each refactoring as an evolution of an API for those aspects of the code that has changed, and also illustrated in this work on API migration in OCaml [6].

5.4 Refactoring schemes

The approach explained in this paper can be seen as a variant of the *refactoring schemes* proposed in [7]. In particular, the examples given in Section 2 instantiate the function refactoring scheme, which can be understood as a strategy that changes function entities in a program by applying rewrite rules to the definition and to the references of the function. When restricting the rewrite rules to only rewrite the name of the function in the reference, the rewrite step does not perform pattern matching and thus it becomes a simple substitution.

6 Conclusions and future work

We have advanced an argument that it is possible to view general refactorings as having a language-independent component, described in the language of function application, and a language-dependent component, materialised by a set of language-specific rewrite rules. This description re-frames earlier work of ours on refactoring for API evolution and language schemes.

We have experimental implementations of the general function refactoring introduced in Section 2 in the Wrangler [14] refactoring tool for Erlang, where it is materialised as an Erlang `behaviour`, and in the Rotor [19] refactoring tool for OCaml. It is a short term goal to finalise and deploy these implementations, as well as articulating the scope and limits of the approach itself.

This work forms part of a longer-term project to build high assurance refactorings. Earlier work on this has concentrated on a formal treatment of (re-)naming in OCaml [18], and a formalisation of the semantics of Erlang [3].

We are very grateful to the referees for their feedback, and in particular their encouragement to contextualise the work more thoroughly, as well as to the ROTOR team at Kent for their insights into refactoring OCaml programs.

References

1 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In Radu Marinescu, Michele Lanza, and Andrian Marcus, editors, *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010. doi:10.1109/ICSM.2010.5610429.

2 Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs Van Der Storm. Study of an API migration for two XML APIs. In *International Conference on Software Language Engineering*, pages 42–61. Springer, 2010.

3 Péter Bereczky, Dániel Horpácsi, and Simon Thompson. A Proof Assistant Based Formalisation of a Subset of Sequential Core Erlang. In Aleksander Byrski and John Hughes, editors, *Trends in Functional Programming*, pages 139–158, Cham, 2020. Springer International Publishing.

4 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72, 2008.

5 Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. QuickChecking Refactoring Tools. In Scott Lystig Fritchie and Konstantinos Sagonas, editors, *Erlang'10: Proceedings of the 2010 ACM SIGPLAN Erlang Workshop*, pages 75–80. ACM SIGPLAN, 2010.

6 Joseph Harrison, Simon Thompson, Steven Varoumas, and Reuben Rowe. API migration: `compare` transformed. In *OCaml Workshop 2020*, 2020.

7 Dániel Horpácsi, Judit Kőszegi, and Zoltán Horváth. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. *Electronic Proceedings in Theoretical Computer Science*, 253:92–108, August 2017. doi:10.4204/eptcs.253.8.

8 Marie-Christine Jakobs. PEQCHECK: Localized and Context-aware Checking of Functional Equivalence. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 130–140, 2021. doi:10.1109/FormaliSE52586.2021.00019.

9 Marie-Christine Jakobs and Maik Wiesner. PEQtest: Testing Functional Equivalence. In Einar Broch Johnsen and Manuel Wimmer, editors, *Fundamental Approaches to Software Engineering*, pages 184–204, Cham, 2022. Springer International Publishing.

10 Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1):9–51, 2004. Special Issue on Program Transformation. doi:doi:10.1016/j.scico.2004.03.002.

11 Ralf Lämmel. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming*, RULE '02, pages 15–28, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/570186.570188.

12 Huiqing Li and Simon Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In M. Di Penta and L. Moonen, editors, *Source Code Analysis and Manipulation, SCAM'06*, 2006.

13 Huiqing Li and Simon Thompson. Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs. In Tim Menzies and Motoshi Saeki, editors, *Automated Software Engineering, ASE'12*. IEEE Computer Society, 2012.

14 Huiqing Li, Simon Thompson, György Orosz, and Melinda Töth. Refactoring with Wrangler, updated. In *ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada*, 2008.

15 Ralf Lämmel and Joost Visser. A Strafunski Application Letter. *Information and Computation/information and Control - IANDC*, pages 357–375, January 2003. doi:10.1007/3-540-36388-2_24.

16 Jeff H. Perkins. Automatically Generating Refactorings to Support API Evolution. *SIGSOFT Softw. Eng. Notes*, 31(1):111–114, September 2005. doi:10.1145/1108768.1108818.

17 Retrie, a powerful, easy-to-use codemodding tool for Haskell., 2020. URL: https://github.com/facebookincubator/retrie.

18 Reuben Rowe, Hugo Férée, Simon Thompson, and Scott Owens. Characterising renaming within Ocaml's module system: theory and implementation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 950–965, 2019.

19 Reuben Rowe, Hugo Férée, Simon Thompson, and Scott Owens. ROTOR: A Tool for Renaming Values in OCaml's Module System. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 27–30, 2019. doi:10.1109/IWoR.2019.00013.

20 Max Schaefer and Oege de Moor. Specifying and Implementing Refactorings. *SIGPLAN Not.*, 45(10):286–301, October 2010. doi:10.1145/1932682.1869485.

21 Hagen Schink, Martin Kuhlemann, Gunter Saake, and Ralf Lämmel. Hurdles in Multi-language Refactoring of Hibernate Applications. In *ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Database Technologies*, volume 2, pages 129–134, January 2011.

22 Friedrich Steimann. Constraint-Based Refactoring. *ACM Trans. Program. Lang. Syst.*, 40(1), January 2018. doi:10.1145/3156016.

23 Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation*. ACM SIGPLAN, 2008.

24 Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring Using Type Constraints. *ACM Trans. Program. Lang. Syst.*, 33(3), May 2011. doi:10.1145/1961204.1961205.

25 Eelco Visser. Language Independent Traversals for Program Transformation. In J. Jeuring, editor, *Proceedings of the Workshop on Generic Programming (WGP2000), Ponte de Lima, Portugal*, July 2000. Technical Report, Department of Information and Computing Sciences, Universiteit Utrecht.