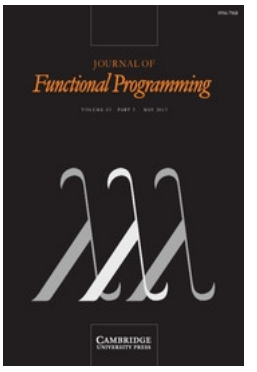

Refactoring tools for functional languages

SIMON THOMPSON and HUIQING LI

Journal of Functional Programming / Volume 23 / Issue 03 / 2013, pp 293 - 350
DOI: 10.1017/S0956796813000117, Published online: 22 October 2013

JFP **23** (3): 293–350, 2013.
doi:10.1017/S0956796813000117

Refactoring tools for functional languages

SIMON THOMPSON and HUIQING LI

School of Computing, University of Kent, Kent, UK
(*e-mail:* {s.j.thompson,h.li}@kent.ac.uk)

Abstract

Refactoring is the process of changing the design of a program without changing what it does. Typical refactorings, such as function extraction and generalisation, are intended to make a program more amenable to extension, more comprehensible and so on. Refactorings differ from other sorts of program transformation in being applied to source code, rather than to a 'core' language within a compiler, and also in having an effect across a code base, rather than to a single function definition, say. Because of this, there is a need to give automated support to the process. This paper reflects on our experience of building tools to refactor functional programs written in Haskell (HaRe) and Erlang (Wrangler). We begin by discussing what refactoring means for functional programming languages, first in theory, and then in the context of a larger example. Next, we address system design and details of system implementation as well as contrasting the style of refactoring and tooling for Haskell and Erlang. Building both tools led to reflections about what particular refactorings mean, as well as requiring analyses of various kinds, and we discuss both of these. We also discuss various extensions to the core tools, including integrating the tools with test frameworks; facilities for detecting and eliminating code clones; and facilities to make the systems extensible by users. We then reflect on our work by drawing some general conclusions, some of which apply particularly to functional languages, while many others are of general value.

1 Introduction

Our aim in this paper is to give an overview of our work in building refactoring tools for Haskell and Erlang, and to reflect on what we have learned and understood in the process, not only about refactoring and refactoring tool building for these two languages but also about refactoring for functional programs in general. In the process we have also made a number of research contributions, and these are summarised too.

The paper will be of value to tool builders and designers, as it reflects on the difficulties and design tradeoffs involved in building practical tools; to language designers, for the reflection on the languages that it provides; to users of the tools, as it will explain not only their design rationale but also what is going on 'under the hood'; to theorists, who can understand the forms of program analysis needed; and finally to researchers in software re-engineering and refactoring in general.

We begin by discussing what is meant by refactoring, and by refactoring for functional programs. This begs the question of whether we can say anything about refactoring for functional programs in general? Our sample of two languages – Haskell and Erlang – shows the breadth of the category 'functional'; here we briefly recall the differences between these

two. Haskell (Marlow, 2010) is a lazy, strongly typed, purely functional language featuring higher-order functions, polymorphism, overloading via type classes and monadic effects. Erlang (Armstrong, 2007) is a strict, weakly typed functional programming language with built-in support for concurrency, communication, distribution, and fault-tolerance. The concurrency primitives and others have side effects. Erlang has an integrated macro language and processor.

Erlang has a single distribution which forms the *de facto* standard; the Haskell standard has been through a number of iterations, but the Glasgow Haskell Compiler (GHC) is itself a *de facto* standard for a superset of Haskell 2010. The standard Erlang distribution comes with a number of libraries, including the Open Telecom Platform (OTP) as well as more specialised libraries, including `syntax_tools`, that gives an abstract interface to the syntax trees produced by the compiler. Haskell is distributed as the Haskell Platform (Haskell Platform, 2010) and this and other packages are available from Hackage (2010), an online repository for open source Haskell projects.

We begin in Section 2 by introducing refactoring and discussing what it means for functional programs in general, introducing along the way a couple of sample refactorings. Section 3 complements this by presenting a larger case study of refactoring a functional program, and discusses which of the refactorings mentioned might be implemented in a tool. Section 4 then gives a high-level introduction to the refactoring tools HaRe and Wrangler, including a brief overview of their design rationale.

Section 5 discusses the implementations of HaRe and Wrangler in more detail, including how they are integrated with various editors and Integrated Development Environments (IDEs); this is followed in Section 6 by an examination of how Wrangler is used in practice. Reflecting on our work, Section 7 discusses a number of design choices we had to make in implementing the refactorings in these tools. The various static analyses which underpin the correctness of the tools are explained in Section 8.

We then turn to various ways in which the tools are enhanced. Section 9 explains how the refactorings implemented in Wrangler are modified to respect the various conventions of testing frameworks for Erlang as well as how refactoring tools can be tested. Section 10 gives an overview of clone detection in Wrangler and HaRe, which gives users reports on the presence of code clones across projects; this is illustrated with case studies on clone detection in an industrial project and the incremental clone detection facilities of Wrangler.

Section 11 describes the reports on problems in module structure as well as other code ‘bad smells’. Section 12 covers the definition of an Application Programming Interface (API) to describe refactorings and a domain-specific language to ‘script’ composite refactorings; the API and Domain-Specific Language (DSL) are illustrated in a practical example. Related work is discussed in Section 13, and in Section 14 we make a number of general reflections on the projects and give an overview of their research contributions. Finally, in Section 15 we draw some conclusions and look at future directions for work in this area.

2 Refactoring functional programs

Refactoring is the process of changing *how* a program achieves a result without changing *what* the result is. Refactorings are transformations to source code, and take place during

program development, evolution and maintenance. The term 'refactoring' was popularised by the book of the same name (Fowler, 1999), but this form of program restructuring has surely gone on ever since programs have been written.

What is distinctive about refactoring functional programs? In order to answer this, we first have to examine in a little more detail about what we mean by 'functional programming' and 'refactoring' here.

Functional programming. The functional programming languages that we have worked with in detail – Haskell and Erlang – have many differences, but at their core is computation by expression evaluation, as exemplified by the λ-calculus. Refactorings at the λ-calculus level consist of expression manipulations modulo $\beta\eta$-equality. An example, which sees the elimination of a common sub-expression followed by an η-contraction, would be:

$$\lambda x.z((\lambda w.yw)x(\lambda w.yw)) = (\lambda p.\lambda x.z(pxp))(\lambda w.yw) = (\lambda p.\lambda x.z(pxp))y$$

However, to be of practical value, a language also needs to have data types, and to have the ability to make definitions, so our working definition of functional programming (for the purpose of the discussion in this section, at least) can be summarised as

$$\text{functional programming} = \text{expressions} + \text{definitions} + \text{data types}$$

Refactorings of a full functional language will affect all three aspects of the language.

Refactoring. In general, refactoring is a program *transformation*, but two other aspects of the process are equally important. Many refactorings have non-trivial *preconditions*; for example, renaming an object of some kind can disrupt the static semantics of the program, and so it is necessary to check that this does not happen if the transformation is indeed to be meaning preserving. These conditions are not simply syntactic, but can depend on static semantics (for binding information), types, module structure and other features; this is discussed in more detail in Section 8.

The need to apply a refactoring may well be indicated by a program '*bad smell*', that is some symptom of the refactoring being both applicable and desirable: smells can range from the local – unnecessary use of the application operator `$` in Haskell, say – to the global – such as the presence of an import cycle in the module graph. Summing this up in a slogan, we can say

$$\text{refactorings} = \text{transformations} + \text{preconditions} + \text{bad smells}.$$

To give a concrete example, consider the process of removing code clones. The 'bad smell' is the presence of duplicate code, and this is eliminated by transforming each clone into a call to a common generalisation; such a transformation is only possible if the generalisation can be defined so that the meanings of the various clones are consistent.[1]

Refactorings are different from the more traditional program transformations that appear in compiler optimisations, for instance. These transformations are usually uni-directional, improving (say) the efficiency of a program, whereas many refactorings are *bi-directional*:

[1] This would include checking that bindings to free identifiers within the bodies are themselves consistent among other conditions; we discuss this further below in Section 8.

	transformation		pre-condition	bad smell
	local	global		
expression	✓		✓	✓
definition	✓	✓	✓	✓
data type		✓	✓	

Fig. 1. Classifying refactorings: ✓primary aspect, ✓secondary aspect.

a `data` type might be transformed into an ADT to allow flexibility in its implementation, whereas the reverse transformation allows pattern matching to be used in definitions. Moreover, because they are transformations of the *source code* of programs, and their results are expected to be intelligible to the programs' authors, their implementation needs to be aware of program layout and commenting.

2.1 Functional refactorings

With these characterisations in mind, we can analyse different kinds of refactorings that are possible for functional programs, as shown in Figure 1, which we discuss in more detail now. We use the black and grey tick symbols to indicate primary and secondary characteristics of refactorings involving different syntactic categories so that 'expression' refactorings are typically local, and characterised by a particular 'bad smell'; there may also be preconditions on their application, but these are less significant in this case. The distinction here between 'local' and 'global' transformations is reflected in the wider literature on refactoring: for instance, Murphy-Hill *et al.* (2009) use the terms 'low-level' and 'high-level' respectively.

Expressions. At this level, the effect of any single transformation will be localised to an expression occurring within a particular definition. These refactorings are often characterised by a 'bad smell', such as bad programming style or verbosity, which is removed by the transformation.

The tools Tidier (Sagonas & Avgerinos, 2009) for Erlang and HLint (Mitchell, 2011) for Haskell implement expression-level refactorings: Tidier implements a set of chosen transformations, fully automatically, whereas HLint serves to point out bad smells that can be eliminated by the user. Feedback from HLint takes the form of advice on the smell and how it can be removed:

```
src\DateTester.hs:50:1: Warning: Use >>=
Found
  do ms <- getMatchers d
     putStr . unlines . map (showMatcher c) $ ms
Why not
  getMatchers d >>= (putStr . unlines . map (showMatcher c))
```

and examples of the application of Tidier can be found in Sagonas & Avgerinos (2009). In the HLint manual Mitchell (2011) discusses the reasons for not applying the transformations automatically, not least of which is the fact that it is difficult to use the existing Haskell

front-end toolset to reformat code so that it looks similar to the untransformed code. On the whole, the preconditions of these transformations are not onerous to compute, and in many cases applying the transformation is simply a matter of matching syntactic patterns.

Definitions and data types. Turning to these, the characteristics of the refactorings are different. The transformations tend to be global: a change in the name of an identifier will have an effect in every module that uses that binding, for instance. Moreover, the preconditions are non-trivial: checking that a renaming is meaning-preserving will depend on a static semantic analysis of each client module for the renamed binding. In a similar way, changing a data type from a concrete representation to an abstract type will affect each use site of data of that type.

To give a concrete example, one of the most useful elementary refactorings is to *generalise* a program item, typically a function, over some specific value 'hard wired' into the definition so that it can be reused more easily. In an example from Erlang, the function `add_one`, which adds one to every element of a list,

```
-module(test).
-export([f/1,add_one/1]).

add_one([H|T]) -> [H+1 | add_one(T)];
add_one([])    -> [].

f(X) -> add_one(X).
```

is generalised over the value `1` to yield this.

```
-module(test).
-export([f/1,add_one/2]).

add_one([H|T],N) -> [H+N | add_one(T,N)];
add_one([],_)    -> [].

f(X) -> add_one(X,1).
```

Observe that the change here is not confined to the definition of `add_one`, but also every call of `add_one` has to be changed too, not only in the `test` module but also in every module that uses the definition. Finally, the list of exports needs to be updated, since arity is significant in identifying a function in Erlang.[2]

Refactorings do not just apply to functions, but can also transform `data` type definitions too. Moreover, a number of simple operations can be used to create a more complex refactoring which transforms a concrete `data` type into an abstract version, as shown in Figure 2. In this case the refactoring is the composition of number of simpler steps, including the introduction of field names for algebraic types and the removal of pattern matching.

[2] Of course, now the function name is misleading, so we need to *rename* it to `add_int` say.

```
module Tree ( Tr(..) ) where

data Tr a = Leaf a
          | Node (Tr a) (Tr a)

flatten :: Tr a -> [a]
flatten (Leaf x) = [x]
flatten (Node s t) = flatten s ++ flatten t
```

```
module Tree (Tr, leaf, left, right, isLeaf, isNode, mkLeaf, mkNode ) where

data Tr a = Leaf {leaf::a}
          | Node {left,right::Tr a}

isLeaf (Leaf _) = True
isLeaf _        = False

mkLeaf = Leaf

isNode = ...
mkNode = ...

flatten :: Tr a -> [a]
flatten t
 | isleaf t = [leaf t]
 | isNode t = flatten (left t) ++ flatten (right t)
```

Fig. 2. A Haskell concrete `data` type of binary trees and its abstract equivalent.

These larger-scale refactorings are less likely to be characterised by clearly defined smells: a name change is appropriate when the effect of (e.g.) a function is not properly reflected by its name, but it is practically impossible to build an automated tool to spot instances like this. On the other hand, some smells like 'code duplication' are only fully detectable by means of automation.

In common with refactoring tools for other languages, in implementing our systems HaRe and Wrangler we have concentrated on larger-scale refactorings, involving structural elements and data types rather than expressions. We have done this because these refactorings are practically impossible without machine support, not only because of the global nature of the transformations but also due to the complexity of checking side conditions in many cases. Thus, our work is complementary to that in Tidier and HLint.

2.2 What is distinctively functional?

So, how is refactoring functional programs different from refactoring for other languages and paradigms?

One difference lies in the *nature of the languages*. Because functional languages are based on expression evaluation, the *expression sub-language* is much richer than in other language paradigms. Tools like Tidier and HLint exploit this, and indeed even without

these tools it is straightforward to implement this kind of transformation 'by hand' as a matter of course during program development.

The *nature of refactorings* can be more general in a functional programming context; we give two examples here. Firstly, the *abstraction* available in the languages allows potentially side-effecting code to be wrapped in a closure: by means of this it is possible to generalise a function over a side-effecting sub-expression in a meaning-preserving way. Not every language supports the definition of closures in this way, whereas it would be unusual for this not to be in a functional language. Secondly, the *higher-order* nature of these languages allows any sub-expression of a function to be the subject of function extraction or generalisation, say, whereas in other languages the types of arguments and results may be limited.

In *implementing* larger-scale refactorings (and indeed smaller-scale too) there are further differences. The semantics of functional languages, although not necessarily fully formally defined, allow full checking of preconditions based on the static semantics of the language in question. They can be decidable and not depend on calculations of control flow, aliasing and so on.[3]

Finally, functional refactorings are potentially more *trustworthy* than for other paradigms. For the pure subsets of languages at least, it is possible to test refactorings by randomly generating programs in the languages, refactoring them randomly, and then testing the results of old and new versions on randomly generated inputs.[4] The relative simplicity of the semantics of the languages makes implementation more straightforward, and also admits the possibility, in principle at least, of proving the implementations of refactorings correct, as discussed in Sultana & Thompson (2008).[5]

Other advantages apply only to particular languages. For example, laziness in Haskell means that the result of a function call is always the same as inlining it: this will only be true in a strict language if the instance of inlining still evaluates the function arguments.

3 Case study: refactoring for program comprehension

In this section we look at a case study of refactoring in practice, where we use refactoring as a mechanism for program comprehension; the particular example is trying to make sense of a student's program to build a semantic tableau system for propositional logic in Haskell.

The purpose of this case study was to discover the kinds of refactorings that it makes sense to automate in designing HaRe as well as to discuss the difficulties of implementing other kinds. The candidates for refactoring are seen to be general (not only applicable in the case study) and atomic (not capable of being broken down into simpler transformations). The case study also demonstrates that machine-supported refactoring is typically only one aspect of refactoring in general because many transformations are only applicable in particular semantic contexts.

[3] In practice, details of languages can undermine this. For example, it is possible in Erlang to convert strings into identifiers, and so identifiers can be computed dynamically, preventing a watertight static analysis. On the other hand, it is possible to identify the (potential) presence of such conversions statically, and to issue appropriate warnings.

[4] Work in this direction is reported in Drienyovszky *et al.* (2010).

[5] This topic is also discussed at the end of Section 14.

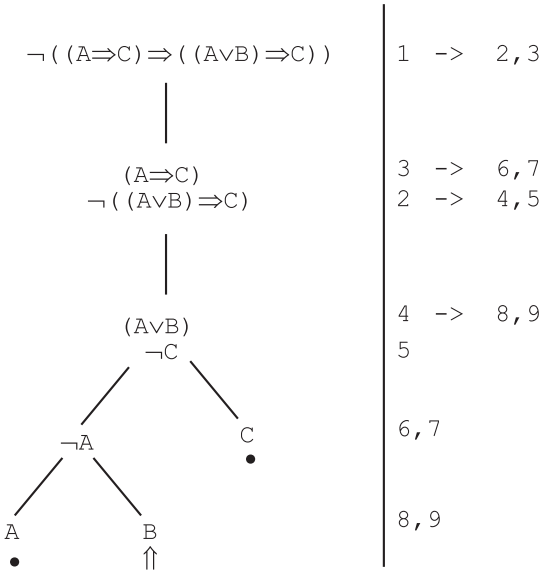

Fig. 3. A semantic tableau for the formula $\neg((A \Rightarrow C) \Rightarrow ((A \vee B) \Rightarrow C)$.

3.1 Semantic tableaux

Semantic tableaux provide a systematic search for all the models of a set of formulas. If no such model can be found, then the set of formulas is unsatisfiable, and in particular if the initial set is $\{\neg\phi\}$ then ϕ is a tautology: semantic tableaux thus provide a decision procedure for validity.

The tableau algorithm operates by successively decomposing formulas. In the case of propositional logic, to satisfy a conjunction, such as $A \wedge B$, it is necessary to satisfy *both* A and B, whereas to satisfy a disjunction $C \vee D$ it is sufficient *either* to satisfy C *or* to satisfy D: this gives rise to a branch point in the tableau.

The tableau in Figure 3 explores all the models for $\neg((A \Rightarrow C) \Rightarrow ((A \vee B) \Rightarrow C))$. The top-level formula is a negated implication $(\neg(\ldots \Rightarrow \ldots))$, which behaves as a conjunction in that it is replaced by the formulas numbered 2 and 3 (as noted on the right-hand side of the figure). By contrast, the decomposition of formula 3, an implication, gives rise to a disjunctive split between $\neg A$ and C (formulas 6 and 7). This decomposition process is repeated until each compound formula has been broken down.

Each branch through the tree represents a potential model, but neither the leftmost nor the rightmost branches corresponds to an actual model, since they each contain both a formula and its negation, both of which cannot be satisfied. On the other hand, the remaining branch does give rise to a model in which $\neg A$, B and $\neg C$ are true, that is a model in which B is true and A and C are false.

3.2 The solution

The program is written in a concrete, first-order style; functions are defined using recursion, and few library functions are used. Propositions are represented by an algebraic type, and the tree is represented by a list of branches, each of which is a list of propositions. This

Filename	LOC	Change from previous version
v0.lhs	401	
v1.lhs	409	Introduce type names: `Branch` for `[Prop]`, `Tableau` for `[[Prop]]`.
v2.lhs	415	Rename functions so that names reflect function behaviour and type, e.g. `removeBranch` is re-named `removeDuplicateBranches`.
v3.hs	418	Change to non-literate form for ease of manipulation.
v4-0.hs	412	Introduce standard HOFs and function-level definitions.
v4-1.hs	419	Correct error introduced at previous stage.
v5-0.hs	394	Change form of 'test' functions to return `Bool` (rather than the argument type), and introduce use of `nub`.
v5-1.hs	416	Correct to use the variant definition of `nub`.
v6.hs	402	Move function to library module.
v7.hs	392	Introduce standard functions, rename `foo` and `bar`, and rename `contra` to `notContra`.
v8.hs	247	Major refactoring to change the flow of control, to allow the merge of functions; requires mitigation of changes.
v9.hs	247	Replace lists by sets in implementation of `Branch` and `Tableau`.

Fig. 4. The refactoring sequence for the tableau case study.

has the advantage of representing the branches explicitly, rather than requiring them to be derived from a different type of tree. The program is a literate script that is 401 lines long, including comments.

The core of any tableau algorithm is an iteration where rules are applied successively until all formulas are expanded; this program applies a single rule to each branch at every iteration. Moreover, the rules are applied in a specified order, which is described in the program by an `Int`; for each branch under consideration this value is calculated and then passed to the iteration function to indicate the rule to be applied.

3.3 The sequence of refactorings

The exercise in program comprehension produced a sequence of twelve versions of the program,[6] and the steps are described in detail in Thompson & Reinke (2003). The steps are summarised in Figure 4, where it can be seen that most stages keep the file approximately the same size if not enlarging it due to the comments added to describe the refactorings.

We can identify three different categories of refactoring in this exercise: Firstly, there is a set of refactorings that are straightforward to describe, and which should be implemented to ensure that they are performed without error, particularly across multiple-module projects.

- The most commonly used refactoring throughout the sequence was *renaming*: some function names were uninformative – including `foo` and `bar` (really!) – or even misleading: `removeBranch` was renamed `removeDuplicateBranches` and `remove` became `removeDuplicatesInBranches`.

[6] The full sequence of refactored versions of the program is available at http://www.cs.kent.ac.uk/projects/refactor-fp/Tableau.zip.

- *Introducing type names*, `Branch` for `[Prop]` and `Tableau` for `[Branch]`, made type declarations easier to read. This also supports changing the implementation type if that is required.
- Some functions visible at the top level were in fact only used by a single definition, and so *demoting the definition* to make it local unclutters the namespace and makes the top-level call graph easier to comprehend.
- Definitions of auxiliary functions were *moved between modules* to separate application code and (more general) library code.

The second class of refactorings replaces a re-definition of a common function by a call to the corresponding library function.

- The programmer used explicit recursive definitions for a number of functions: these could be *replaced with library calls* in many cases. These are calls to both *higher-order functions*, including `map` and `filter`, and *first-order* operations such as `concat` and `elem`.
  This process can introduce subtle errors: for example, the library version of `nub` preserves the *first* occurrence of each element, whereas the programmer had written a version preserving the *last*. Since the particular algorithm was sensitive to the ordering of elements in the lists, this led to the program failing in some cases.

However, it is more problematic to implement this set because of the number of different ways it is possible to write equivalent definitions, particularly in Haskell. For example, in this particular case study, the definition of a mapping function in the program used `[x]++xs` instead of `x:xs` for list construction; moreover, in a number of cases `[]++ys` is used in place of `ys` (perhaps to emphasise symmetry with other cases). It is therefore difficult completely to automate the process of recognising redefined library functions.

Finally, among the refactorings are larger-scale transformations which depend on the semantics of the particular program for their correctness.

- A particular programming style used in the implementation disguised tests (`f` say) over a particular type (e.g. `Prop`) as functions that return a `Prop`, but which are only called in contexts of the form f x == x. These can be transformed into Boolean functions, provided that the function is not called on the value of `Prop` used to represent `False`.
- The particular design of the program consisted of functions `splitX`, `removeX` and `solveX` for a `X` running through a number of combinations – one per rule, effectively. This results in code duplication across the different instantiations. This was refactored into three functions `split`, `remove` and `solve`, but needed further changes (e.g. list sorting) to alleviate changes due to the sensitivity of the algorithm to list (and list of list) ordering.
- It was always the case that the problem itself was based on sets rather than lists, and as a final refactoring this was achieved, replacing functions in a list API with the corresponding functions for sets (implemented as lists). The correctness of this depends on particular algebraic properties of operations used over the sets, e.g. commutativity and associativity of operations folded over a set.

It would not make sense to implement these refactorings within a tool for a number of reasons. Firstly, their sensitivity to the particular semantics of the program means that their preconditions would be difficult if not impossible to define. Secondly, they are unlikely to be useful in general, because they come out of a very specific scenario. However, it might be possible for programmers to define refactorings of this kind using the extension facilities discussed in Sections 12.1 and 12.2 below.

4 Systems

In this section we give an overview of the two refactoring tools, HaRe and Wrangler, before describing their implementation in the next section.

4.1 HaRe

HaRe (HaRe, n.d.; Li, 2006; Brown, 2008) provides support for refactorings and covers the full Haskell 98 standard (Hughes & Peyton Jones, 1999). It covers a range of structural refactorings, module refactorings and data-type oriented refactorings.

Structural refactorings mainly concern the name and scope of the entities defined in a program and the structure of definition. Structural refactorings supported by HaRe include

- renaming of variable, function and modules;
- deleting a definition that is not used;
- duplicating a function definition under a user-provided new name;
- promoting a definition from a local scope to a wider scope;
- demoting a definition which is only used within one definition to be local to that definition;
- unfolding a definition by replacing an identified occurrence of the left-hand side of a definition with the instantiated right-hand side;
- folding that replaces sub-expressions which are substitution instances of the right-hand side of an identified definition with a call to that definition;
- *generative folding* as described in Burstall & Darlington (1975);
- folding/unfolding as patterns;
- conversion between a `let` expressions and a `where` clause;
- introducing a new definition to name a user-identified expression;
- generalisation of a function definition, as shown in Section 2.1;
- adding/removing an argument to/from a function definition; and
- simplification of case expressions.

Module refactorings concern the imports and exports of individual modules, and the relocation of definitions among modules, and these refactorings include

- cleaning an import list to remove redundant import declarations and entities;
- adding to an import declaration an explicit list of all the imported entities that are actually used by the module;
- adding an entity to the export list of a module;
- removing an entity from the export list; and
- moving a definition from one module to another.

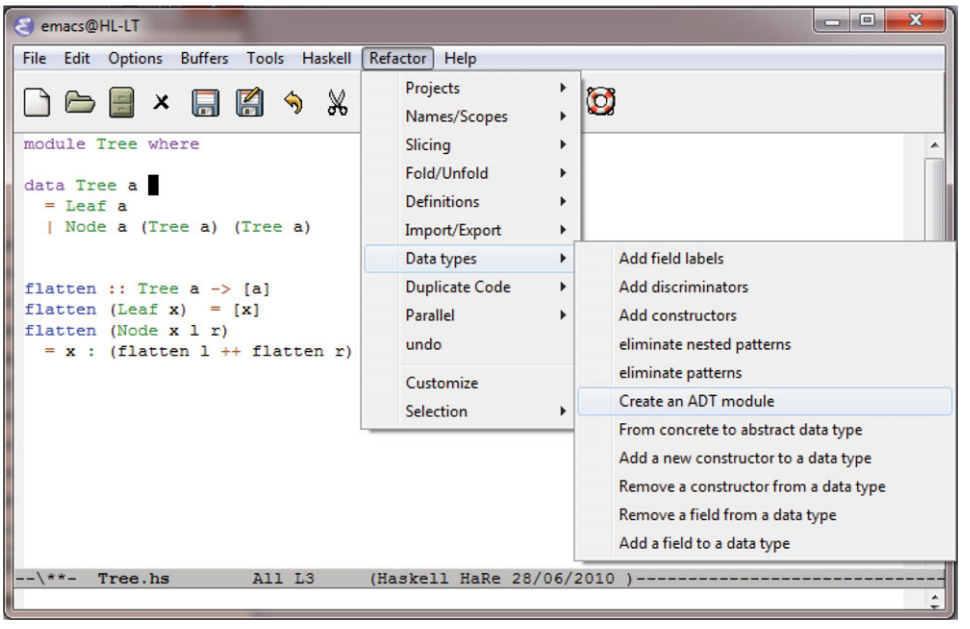

Fig. 5. (Colour online) Screenshot of HaRe within Windows Emacs.

Data-type refactorings affect the type definitions of a program, and also definitions, or expressions, that make use of a type definition being refactored. Data-type-related refactorings supported by HaRe include

- from concrete to abstract data type, as shown in Figure 2, is a composite refactoring consisting of a number of elementary refactoring steps;
- adding/removing a constructor to/from data-type definition;
- adding/removing a field to/from an identified data-type constructor; and
- introducing pattern matching over an argument of a function definition.

Apart from the various refactorings, HaRe also has some extensions, including the functionalities for detecting and eliminating similar code (or 'code clones') (Brown & Thompson, 2010) as well as functionalities for program slicing. HaRe has recently been extended to include refactorings for the parallelisation of Haskell programs (Brown *et al.*, 2011).

HaRe is integrated with the editors Vim and (X) Emacs: see Figure 5. HaRe is available from Hackage, as well as being downloadable from http://github.com/RefactoringTools.

4.2 Wrangler

Wrangler (Li & Thompson, 2006; Li *et al.*, 2008) provides a set of structural refactorings similar to those in HaRe. Specifically they include

- renaming of variables, functions and modules;
- function generalisation, as shown above;

- function extraction: an identified expression is made for the body of a new function definition;
- inlining or 'unfolding' a function application, and its dual, 'folding' an expression into a function application;
- introduction of local variables to name an identified expression, and dually, to inline variable definitions; and
- introduction of new macro definitions and to fold against macro definitions.

These refactorings are similar to those in HaRe, but are tuned to Erlang syntax and semantics rather than Haskell; to take two examples,

- since Erlang expressions can have side effects, it is sometimes necessary to wrap the argument to a generalised function in a closure;
- local functions can be introduced: not directly as local definitions (which do not feature in Erlang) but as functional values stored in local variables.

Wrangler also provides a set of process-based refactorings, including the introduction of process naming, instead of providing access to a process only through a dynamically generated 'process id'. Wrangler also contains a variety of refactorings which work with Erlang QuickCheck, introducing and merging constructs such as `?LET` and `?FORALL`.

Transforming data types in Erlang can be difficult, as Erlang functions can take values of any type, in principle. There are situations – particularly in implementing an Erlang *behaviour*, which can be thought of as an interface for a set of callback functions – where the types are more constrained, and where data transformations are possible. In particular, we have implemented transformations for *state data* in various kinds of generic state machines in both the OTP library and the QuickCheck.

In implementing Wrangler we have chosen to respect various features of the language and related tools. Wrangler is able to process modules which use *macros*, including the Erlang test frameworks that are in regular use. Wrangler also respects the naming conventions in those test frameworks (Li & Thompson, 2009b).

Wrangler provides a portfolio of decision support tools. The code inspector highlights local 'code smells', and a number of reports highlight issues in the module structure of projects, including circular inclusions and other potential faults (Li & Thompson, 2010a). The code clone detection facilities can be used on large multi-module projects to report on code clones and how they can be removed; clone detection can be preformed incrementally on larger code bases, for example, as part of a continuous integration approach to software construction (Li & Thompson, 2009a, 2011b.

Wrangler is integrated within Emacs – including XEmacs – and also within Eclipse as a part of the Erlang IDE or ErlIDE (ErlIDE, n.d.) plugin for Erlang. A screenshot of the embedding within the Mac OS X variant of Emacs, Aquamacs, is shown in Figure 6. The Emacs and Eclipse versions provide a preview of the effects of a refactoring and also support multi-level 'undo' once refactorings have been performed.

Wrangler has been recently extended with a framework that allows users to define for themselves refactorings and code inspection functions that suit their needs (Li & Thompson, 2011a). These are defined using a template- and rule-based program transformation and analysis API. Wrangler also supports a domain-specific language that allows users to script composite refactorings, test them and apply them on the fly (Li & Thompson,

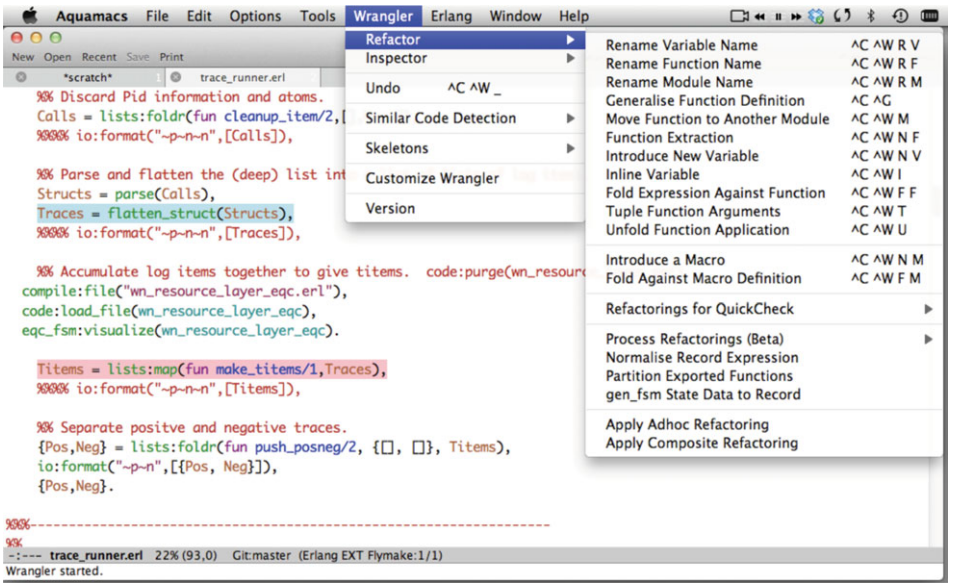

Fig. 6. (Colour online) Screenshot of wrangler within Aquamacs Emacs.

2012a). User-defined refactorings and scripts are not 'second-class citizens': like the existing Wrangler refactorings, user-defined refactorings benefit from features such as results preview, layout preservation, selective refactoring, undo and so on.

4.3 Design rationale

A major aim in building these tools was to make them attractive to users. The top-level design of these refactoring tools was therefore guided by a number of principles, informed by discussions with practitioners; we discuss these briefly here.

Target the full language. Working with a language subset can provide a 'proof of concept' but only a complete implementation will be usable in practice. For this reason, HaRe works with Haskell 98 and Wrangler with Erlang/OTP releases R11 onwards (currently R16; see also the discussion of past and present obstacles in Section 14).

Produce readable results. The code that is produced needs not only to be comprehensible but should also look like the original code. We therefore need to preserve code layout as much as possible, and also to ensure that any new code that is generated has a similar appearance too. We also need to ensure that comments are preserved (or indeed refactored!) and moved with code when it is migrated.

Workflow integration. The system should be integrated with the user's usual workflow, and so we have chosen to integrate with appropriate environments, including Vim, Emacs and Eclipse. Each system is implemented in the host language, and refactoring commands are simply function calls. This allows refactorings to be invoked from the command line, and also supports the integration of the tools in other editors and IDEs.

Support incomplete code. Sometimes we want to refactor code that does not (fully) compile. For example, in Wrangler we can parse modules function by function, and so rename a function that parses, irrespective of parse errors elsewhere in the module.[7]

Support user decisions. Users wish to understand the effect of a refactoring before committing it: this is done by supporting a preview option, and also allowing refactoring steps to be 'undone'. Users are also keen to get decision-support assistance in finding potential refactorings and in applying them: we address this in Sections 10 and 11.

Support user-driven extension. It is impossible to provide all the facilities that users might require, and so in Wrangler we provide the facility for users to write new refactorings (Section 12.1) and a DSL for scripting complex refactorings (Section 12.2). The tools are also open source, and have attracted external contributions.

Integrate with other tools. Program code does not exist in a vacuum, but is part of a wider system of tools, such as configuration and build systems, documentation and testing tools. Wrangler has been designed to support integration with test frameworks (Section 9) and provides a set of hooks into source code repositories. In general, integration with tools is made easier in an IDE, which may well integrate the tools already, rather than in an editor like Emacs.

5 Implementation

The high-level architecture of HaRe and Wrangler is the same, and forms the skeleton of the diagrams in Figure 7. Program text is parsed into an abstract syntax tree, which is then augmented with additional semantic information. Walking this tree allows the computation of the preconditions for the refactoring, and if these conditions hold then the tree is transformed; it is finally rendered back into program source code.

5.1 General approach

Each tool is implemented in its own language: HaRe in Haskell and Wrangler in Erlang; this allows us to take advantage of existing toolkits and infrastructure written in each language. Refactoring support is supplied by a separate process performing the refactorings: these are initiated by messages sent from the host editor, and performed on the file system. Refactoring interactions and so forth are provided by the host editor, using whichever language that editor supports: Elisp in the case of Emacs, and Java within Eclipse.

Programs are represented internally within a refactoring tool by abstract syntax trees, augmented with additional information, including static semantics, type information, module information and so forth. The parse trees can be supplied by a parser for the language, and other information can be gathered from the front end of a compiler for the language, or computed by the tool itself.

Information about program semantics, data to use in precondition checking and also the program transformations are achieved by walking the trees that represent the abstract syntax of the program. Typical of the operations is a generic tree walk, which is the identity

[7] There is a question about whether this transformation preserves meaning, but we claim that it does since an unparsable function must be formally meaningless. Wrangler also generates a warning message about the presence of any parse errors.

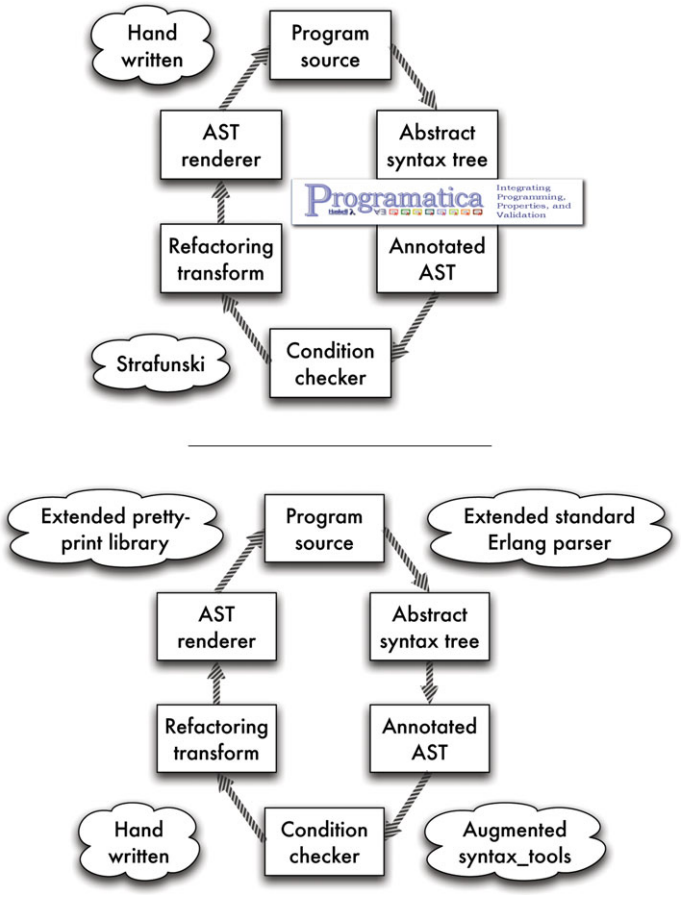

Fig. 7. (Colour online) The architecture of HaRe (above) and Wrangler (below).

except for different *ad hoc* behaviour at particular types or constructors. For instance, a function renaming will affect only those tree nodes which reference the function being renamed. Traversals of this form are easily accomplished using *strategic programming* in the style of Stratego (Bravenboer *et al.*, 2008), which provides an untyped approach which carries across directly into Erlang. Haskell currently supports a plethora of different approaches to generic programming; at the time of design, Strafunski (Lämmel & Visser, 2003) gave the best support for this style of traversal in a typed environment.

Finally, the modified syntax tree needs to be rendered back into code: formatting this can be guided by layout information in the tree itself (or within the token stream), but it is also necessary to pretty print parts of the code which are synthesised by the refactoring transformation.

While the top-level designs of the two systems are similar, they differ in a number of important details; we examine these now.

5.2 HaRe

HaRe was designed to reuse existing systems whenever possible, and when it was designed in 2002 the most appropriate choices were to use Programatica (Hallgren, 2003) for the

front-end processing and Strafunski (Lämmel & Visser, 2003) to support precondition checking and transformation. For efficiency reason, we used the type checker from GHC, instead of Programatica, to derive type information. In HaRe, we use both the AST and the token stream as the internal representation of source code. Layout and comment information is kept in the token stream, and some layout information is kept in the AST. The refactorer carries out program analysis with the AST, but performs program transformation on both the AST and the token stream, that is whenever the AST is modified, the token stream will also be modified to reflect the changes. After a refactoring, the new source code is extracted from the transformed token stream.

5.3 Wrangler

Wrangler is built on top of the `syntax_tools` library from the Erlang/OTP release. This library provides an extended version of the standard Erlang parser that accepts pre-processor directives and macro applications as well as an abstract interface to the syntax trees produced by the parser. In addition to that, `syntax_tools` also provides functionalities for reading comments from Erlang source code and for inserting comments as attachments to AST nodes at correct places as well as the functionality for pretty-printing of Erlang ASTs decorated with comments. Traversing an Erlang AST generated by `syntax_tools` is straightforward since all the non-leaf nodes in the AST have the same type.

We have extended the `syntax_tools` library with functionalities for adding static semantic and location information to the AST. For example, the binding structure of identifiers is stored in the AST by annotating each identifier occurrence with its defining location; each node in the AST is annotated with its start and end locations within the program source in terms of line and column numbers as well as its syntax context information etc.

Wrangler's layout preservation is achieved in a different way from HaRe. Instead of consistent updating of both the AST and the token stream, Wrangler only updates the AST. A modified version of the pretty-printer is used to layout the code after a refactoring. The pretty-printer is modified in such a way that the original program layout information is used to guide the pretty-printing process whenever possible.

With Erlang as the implementation language of Wrangler, reusing and incrementally updating existing information are easily achievable using Erlang concurrent processes, and that is the approach we have adopted. For example, a `gen_server` process, called AST server, is dedicated to the AST management. If the AST of an Erlang file is needed, the refactorer will ask the AST server for it. Within the AST server, an Erlang module is parsed only when its AST does not exist or is out of date. The refactorer also informs the AST server whenever a module has been refactored, which will then update its AST repository in the background. In a similar way, there is also an Erlang process in charge of maintaining the module/function callgraph in the background.

5.4 Describing refactorings

Communication between the front end (Emacs etc.) and the refactoring server uses a textual protocol to describe particular *refactoring commands*. A typical command is described by a combination of the following elements:

refactoring: The name of the refactoring to be performed.
paths: Details of the paths describing the scope of the project under refactoring.
names: A new name for a function or module, for instance.
focus: The current focus in the editor, a single position, determines the object of a refactoring, e.g. the function to be renamed.
selection: The current selection, which is described by a pair comprising the start and end positions, is needed to describe the object of other refactorings, such as function extraction.
choices: In the case where choices are gathered interactively, these results are collected into a list of Y/y/N/n values.
tab key: The number of spaces per tab key, which is necessary for the tool to resolve locations correctly.

This form of description requires the least computation within the front end so that, for example, the resolution of the cursor position with the AST is performed at the server.

While this interface works well internally, other options provide a better API to the programmer. Rather than using the position within a text file, the position of the object of a refactoring could be indicated by name, or by a path within the AST, for instance. Textual position does not provide a robust description of position, since it will be changed by applying a refactoring transformation. The alternatives make it easier to describe a series of refactorings, since the logical position or name of an object will be unchanged, or at least will be changed in a more predictable way than textual position. This increased robustness makes it easier for series of refactorings to be scripted.

Currently, a refactoring is applied by directly modifying the source files affected. We can then extract the differences between the old and new files to show the effect of a transformation. Alternatively, our implementation could generate diffs or patches directly, and these could then be applied to the project files. This approach might have the advantage that commuting the application of refactorings could more easily be supported in this model than under the 'change the whole file' approach.

A higher-level approach to describe refactorings is provided by the API and the DSL provided by Wrangler; these are described in more detail in Sections 12.1 and 12.2.

5.5 Workflow integration

Workflow integration means that programmers can use intrinsicthese refactoring tools without leaving their favourite editor. Wrangler and HaRe are integrated into Emacs, Eclipse and Vim through their mechanisms for communication with externally running processes. Each is more than just an editor, and particularly Emacs and Eclipse provide plugin functionality to perform a variety of tasks, including showing file diffs, displaying graphs in 'dot' (graphviz) format as well as the integration with version control systems.

Emacs and Vim. Gnu Emacs is programmable in Elisp (Chassell, 2004) and this makes extension relatively straightforward. It is portable across different OS platforms, but with variants that localise, e.g. Aquamacs for Mac OS X. Emacs has well-developed modes for Haskell and Erlang, and also provides front end functionality for other tools such as Quviq QuickCheck. We are able to use the `ediff` plugin to show the diffs between the file before

and after a refactoring, and we can provide reports with links back into the code by using compilation-minor mode.

Emacs has no intrinsic notion of a project (although the Emacs Development Environment, EDE, add-on provides some IDE-like extensions), and so it only 'knows about' the files that it is currently editing. This is a problem if we wish to undo the effect of a project-wide refactoring, since this can then only be done by putting the 'undo' functionality in the refactorer, rather than embedding it within Emacs.

By contrast, it is more difficult to program extensions to Vim, and some aspects of Vim are not platform-independent: for example, the mechanism for calling external processes is different on Windows and Unix systems. It also shares the disadvantages of Emacs *vis à vis* Eclipse, which we discuss now.

Eclipse. Eclipse is a fully featured programming IDE written in Java, providing support for development in Java and a number of other programming languages. We build on top of ErlIDE (ErlIDE, n.d.), an Erlang plugin for Eclipse.

Eclipse is programmable in Java, and moreover provides an interface for refactorings to implement. Refactorings that meet this API will be treated as refactorings by Eclipse so that they appear in the menu that users expect, they automatically have preview functionality, their 'undo' is integrated with the Eclipse 'undo' and so forth (Li *et al.*, 2008). The downside of this is that the workflow these refactorings are expected to follow is limited and required modification of our original implementations to achieve.

Eclipse brings three other advantages. It has a notion of project which naturally delimits the scope of a refactoring; this also means that there are stronger links with the testing and build processes than within Emacs. Because it is a graphical IDE, choices (e.g. of which instances to refactor) can be made in a declarative way by presenting the user with a list of all instances from which the user can select a subset by direct manipulation, rather than through an interactive dialogue. Finally, Eclipse serves a quite different set of developers than Emacs, and those coming to Erlang or Haskell from the Ruby or .NET communities, for example, will expect good IDE support.

6 Case study: wrangler in action

Figure 8 presents information about how Wrangler has been used in practice in two case studies during the period January 2010 to July 2011. Data for these were gained by instrumenting the system to log refactoring calls and their results.The first data come from the use by the Wrangler team on Wrangler itself, and the second come from a staff team at LambdaStream on an Erlang-based product. We can see some similarities and some differences in the data.

- Function extraction – in which a highlighted code fragment becomes the body of a new function – is the most used refactoring in the LambdaStream study, while it is heavily used in Wrangler too, it is only the fourth most popular. Use of this can be accompanied by folding against the newly introduced function, and the data do not contradict that.
- We see generalisation in use in both case studies: This can be for itself, or can contribute to a higher-level refactoring, such as (manual) clone extraction and removal.

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	
TOTAL	1246	158

Fig. 8. Wrangler refactoring usage in two case studies.

- Simple structural refactorings – renaming variables and functions, and moving functions between modules – are the most used in the Wrangler study. We attribute this to the use of Wrangler on itself to tidy up the system periodically, while the principal use of Wrangler at LambdaStream was during development, when function extraction can contribute directly to the code being written.
- Finally, we note that module renaming was used heavily in Wrangler. This is impractical for refactoring 'by hand' in Erlang, since calls to functions in external modules contain the external module name, and so require changes throughout the code base. Automation makes it feasible.

Efficiency. Refactoring tools are supposed to be used in an interactive way, therefore the response time should be short enough to be bearable for users. Various techniques have been used in the tools to reuse and incrementally update existing information such as the AST, module graph, call graph etc. For typical refactoring tasks performed during interactive program development, the system is able to provide adequate interactive responses within a small number of seconds.

Looking at larger-scale refactorings, which would only be undertaken with some thought, in a typical example it takes Wrangler 51 seconds to rename the standard module `lists` to some other name across the Erlang standard library `stdlib`, which contains 78 Erlang files and 72.9k lines of code. This refactoring not only renames the module itself but also all of the references in this module across the `stdlib` library. As a result of this refactoring, 59 out of the 78 Erlang modules are changed. The 51 seconds discussed here applies when all the modules are already loaded into the system; the first run during which all modules are parsed, analysed and stored in the AST server for future use takes 123 seconds.

As a final example, it takes 36 seconds (94 seconds for the first run) to use Wrangler's incremental clone detection facility to report 78 sets of clones from a codebase, of size

244.4k lines of code, including the application and testing code of both the Erlang compiler and the Erlang `stdlib`.[8]

7 Design experience

Implementing the tools led us to see that there are often *design choices* to be made in the way that particular refactorings are implemented. These choices reflect general options which the implementer needs to resolve. We discuss four examples in more detail in this section.

7.1 What do you mean?

Suppose that we implement a refactoring to generalise a definition over a sub-expression occurring within the body of the definition. As a concrete example, suppose we are to generalise the following definition over the indicated sub-expression, namely `1`.

```
-module (setup).
-export([port/1]).

port() ->
  PortId           = 1,
  SessionId        = 127+1,
  Version          = 1,
  {PortId,SessionId,Version}.
```

Three eventualities suggest themselves:

Single occurrence. We can generalise over the single indicated occurrence, giving:

```
port(N) ->
  PortId    = 1,
  SessionId = 127+N,
  Version   = 1,
  {PortId,SessionId,Version}.
```

All occurrences. Dually, we could generalise over all occurrences of `1`, with the result:

```
port(N) ->
  PortId    = N,
  SessionId = 127+N,
  Version   = N,
  {PortId,SessionId,Version}.
```

Some occurrences. Finally, we might want to have the choice of generalising over a selection of the occurrences, as shown here:

[8] These measurements were run on a laptop with a 2.27-GHz Intel(R) processor, 4-GB RAM and running Windows 7.

```
port() ->
  PortId    = 1,
  SessionId = 127+[1],
  Version   = [1],
  {PortId,SessionId,Version}.
```

This choice between *one, all and some* is repeated in a number of other situations, and tools need to be designed to reflect this. Within an editor like Emacs or Vim, we step interactively through the options, offering the user a chance at each stage to choose yes or no to the current option, and also to all the remaining options.

Within an IDE with a graphical user interface we can open a window, within which we can present a clickable list of (links to) all the occurrences of the clone. A user can then select a subset of these by clicking, thus giving them a chance to survey all the occurrences before making a choice rather than having to make a sequence of decisions instance by instance. A choice like this can be complicated in a number of ways.

Polymorphism. Given a Haskell definition of the form

```
f :: (Num t) => [t] -> [Bool] -> [t] -> Int

f x y z = length ((2:x) ++ [[]]) +
          length ((True:y) ++ []) +
          length ((3:z) ++ [])
```

suppose that the first empty list is selected as the expression over which to generalise. We still have options here, since we may also generalise over the third occurrence of `[]`, but since arguments to Haskell (Haskell 2010) functions cannot be forced to be polymorphic, it is not possible to generalise over the second `[]` since it is an empty list of Booleans rather than numbers, and generalising over both occurrences would force the argument to be of type $\forall$`a.[a]`.

To conclude, the set of possible occurrences is governed not just by the expression but also by its type. Any refactoring tool will therefore need to be *type aware*.

Multiple arities. Erlang functions are determined not just by their name but also by their arity, and so it is not unusual to see Erlang functions with the same name but different arity: a common example is the `reverse/1` function where `reverse/2` is the tail-recursive auxiliary function which does the work of shunting elements from one list to another. In generalising

```
reply(start) ->
  dest ! {self(),start}
reply(skip) ->
  ok.
```

over the destination `dest` we have a choice: we could just generalise the first clause, or both the clauses. The first option would give rise to functions `reply/2` and `reply/1`, and the second just to `reply/2`.

7.2 Compensate or reject?

Suppose we want to lift the local definition of g in

```
h x = x + g x
    where
      g x = x + con
      con = 37
```

to the top level. Since g depends on the local definition of con, the simplest option is to *reject* this attempted refactoring, and to expect the user to take the appropriate remedial action before the definition can be lifted.

The alternative is to perform some *automatic compensation* to allow the refactoring to take place. In this particular case there are (at least) four distinct options:

- We can *lambda lift* the definition so that con becomes a formal parameter to the definition with the actual value being passed in at the call site. This maintains the local definition of con in its original scope,

  ```
  h x = x + g con x
      where
        con = 37

  g con x = x + con
  ```

- We could also argue that as well as lifting g we should also lift all its dependents so that here both con and g are lifted. This changes more scopes, but preserves the arity of g in contrast to the lambda-lifted version,

  ```
  h x = x + g x

  g x = x + con
  con = 37
  ```

- In this particular example there is a third possibility that only adds g to the top-level scope, and also preserves its arity: we first make con local to g before lifting the definition,

  ```
  h x = x + g x

  g x = x + con
      where
        con = 37
  ```

  Since con is only used in g, it can be removed from h; in the general case it would have to remain as a local definition to h as well as being added to g.

- Finally, we could *unfold* the definition of con before lifting g, giving

  ```
  h x = x + g x

  g x = x + 37
  ```

In making a choice of how to implement this lifting operation, there is a tension between the simplicity of making no compensation and the greater ease of use that some compensation gives. We have chosen to lambda lift in this case, but we note that this does not preclude users from performing other compensations manually should they so wish.

7.3 Backwards compatibility?

Suppose that we are to generalise the Erlang function `add_one` from the `test` module over the expression `1`.

```
-module (test).
-export([add_one/1]).

add_one([H|T]) -> [H+1 | add_one(T)];
add_one([])    -> [].
```

Our refactoring tools are able to survey all the modules in the project to hand, and to modify all calls `add_one(L)` to the corresponding calls to `add_one(L,1)`.

The closed world assumption – that we have access to all calls to `add_one/1` – is maybe not tenable. Some modules may only be available in binary form, or clients may use our code without disclosing how it is used. So there is a question of whether we should include a *legacy* version of `add_one`, as shown by the *slanted* code here:

```
-module (test).
-export([add_one/1,add_one/2]).

add_one([H|T],N) -> [H+N | add_one(T,N)];
add_one([],N)    -> [].

add_one(L)       -> add_one(L,1).
```

The argument for doing this is that it preserves code compilability; the disadvantage is the invisibility of this compensation. Arguably, the best option is to raise a warning when legacy calls are made, but the Erlang philosophy would simply be to 'let it fail' and for the problem to be fixed at the point of failure.

Another case to consider is the use of the Erlang primitive built-in function (or BIF) `list_to_atom` which turns a text string into an atom. Since module and function identifiers are themselves atoms, this means that identifiers can be created dynamically. While this sort of programming might be discouraged in general, it is not uncommon in *meta-programming* examples. Since names are created dynamically by this BIF, it is impossible to detect instances of names generated by this BIF when, for instance, a renaming refactoring is performed. One form of mitigation would be to wrap all calls to `list_to_atom` to check and dynamically apply refactorings as appropriate. Our approach has been to warn the programmer about uses of `list_to_atom`, but not to take any mitigating action, and instead to leave the program to fail under such circumstances.

7.4 90% or nothing?

In some situations – such as the data-flow analysis of variables in Section 8.6 – it may well not be possible to decide all cases where an action – replace a Pid by a name, say – should take place. In this particular example we are trying to replace a dynamic artefact by a static one, and so there are fundamental obstacles to performing all and only the required changes.

One option would be to leave the system unchanged, because we may not be exhaustive; the other option is to warn the user that there might be cases that she should look at, and for the system to make a conservative estimate of these. This is precisely what we do in the running example, since it delivers a transformation that is almost complete, and points the user to ways of completing it, rather than simply 'bottling out' for reasons of correctness. The point here is that our pragmatic choice is more useful than providing nothing and expecting the user to complete the transformation by hand herself.

8 Analysis

Program analyses of various kinds are needed not only for computing preconditions of refactorings but also for effecting refactoring transformations themselves. This section surveys the different analyses that are used.

8.1 Static semantics

It goes without saying that it is necessary to use the static semantics of the language to resolve bindings of identifiers, since this sort of analysis underlies the conditions and implementation of almost all refactorings. A renaming refactoring should not affect the binding structure of a program, while function extraction and generalisation should also preserve the bindings in code that is moved in some way within the AST.

It is interesting to note how the binding structures of Haskell and Erlang differ in a number of small but substantial ways. In Haskell any occurrence of a bound variable in a *pattern* is a binding occurrence, whereas in Erlang bound variables can be used in patterns, as in the example of `Pid` here.

```
receiveFrom(Pid) ->
  receive
    {Pid,Payload} -> ...
    ... -> ...
  end.
```

Erlang also allows multiple binding occurrences of a variable: for example, in all the arms of a conditional statement, as here for `X`,

```
foo(Z) ->
  case Z of
    {foo,Foo} -> X=37;
    {bar,Bar} -> X=42
  end,
  X+1.
```

The subtle differences that we see between the two languages make it difficult to imagine that a fully generic tool for refactoring functional programs could be built; we take up this question again in Section 14.

8.2 Types

Many refactorings require a type analysis to be performed. A Haskell example is given by the generalisation of the function `process` over the indicated value `[]`,

```
process xs =
  zip (True:xs ++ [], [1,2,3]++[])
```

The empty list has a polymorphic type, and in this example its first use is as a list of booleans, while its second use is as a numeric list. It is therefore not possible to generalise over both the lists at once, since this would force the new argument to be polymorphic, and this is not permitted in Haskell 98. So a simple syntactic analysis is not sufficient here, and it is necessary to know the types of expressions when performing generalisation. A related example is given by

```
quantize x = round (x / 10) * 10
```

which when generalised over both instances of `10` gives the function

```
quantize2 :: (RealFrac a, Integral a) => a -> a -> a
quantize2 y x = round (x / y) * y
```

There is no numeric type that is an instance of both the classes, `RealFrac` and `Integral`, and so this function cannot in fact be applied to any concrete numeric arguments.

Many Haskell programmers include type declarations for all their definitions as a matter of course, and these need to be refactored too. One mechanism for doing this is to use type inference to infer the type of the new function, for instance, after generalisation. This will give a valid typing for a function, but may not be ideal. It will not involve any type synonyms which were used in the original declaration, and it will also give the most general type which may have been overridden to something more specific by the original type declaration. For these reasons it is preferable to use the type of the new parameter together with the original declaration to produce the type of the generalised function. Type inference may also fail in the case of polymorphic recursion, thus requiring type annotation.

Erlang is a weakly typed language, but types do play a role in refactorings. For instance, Erlang supports record types, which are implemented by syntactic sugar based on tuples, and records are often introduced to replace tuples. A typical function manipulating a tuple

```
foo({Pid,Payload}) -> Payload+1.
```

would be replaced by this record syntax

```
foo(Z) -> Z#msg.payload+1.
```

but because Erlang types are weak, it is not clear whether this is a sufficient replacement, or whether the original, tuple-manipulating definition needs to be retained to catch any instances which had not been transformed.

8.3 Modules

Our refactorings are designed to work across projects, rather than simply to work over single files. Eclipse has a notion of project, but for other front ends, we need to specify projects by sets of directories in which code will belong. If the focus of a refactoring is in module `M` say, then it is necessary to know the modules on which `M` depends – for instance, to gather type or side effect information – and also the modules which depend on `M`, as these are the modules that will potentially change because of a refactoring.

Information about dependencies is conveyed in different ways in different languages. In Haskell it is necessary to `import` modules in order to use definitions,

```
module Server(processMsg) where
import Messaging
processMsg z = format(msg(z))
```

While import is possible in Erlang, the strongly held convention is that external calls should be fully qualified, and so explicitly indicate the module to which they refer,

```
-module(server).
-export([processMsg/1]).
processMsg(Z) ->
  format(messaging:msg(Z)).
```

Module graph and call graph information remains relatively stable across refactorings, and so it makes sense for this data to persist. The Erlang concurrency model makes this particularly easy to achieve: we spawn a 'module graph server' which maintains information during a refactoring session, updating the dependency graph only when a potential change has occurred.

8.4 Side effects

Side effects in Haskell code are handled by wrapping them in monadic computations, and so these effects only happen when a computation is explicitly run. On the other hand, any Erlang expression can potentially cause a side effect, and this impacts on some refactorings. For example, naively generalising over an expression that has side effects will not be sound. Consider the example,

```
printList(0) -> true;
printList(N) ->
  io:format("*"),
  printList(N-1).

example() -> printlist(3).
```

If we simply pass in the expression `io:format("*")` as an argument, then a single star will be printed when the argument is evaluated, rather than when it should. The solution for this is to wrap the side-effecting expression up as a *closure*, like this:

```
printList(F,0) -> true;
printList(F,N) ->
  F(),
  printList(F,N-1).

example() ->
  printlist(fun()->io:format("*") end,3).
```

It would be possible to wrap *all* generalisation parameters as closures, but this is unnecessarily defensive and also obscures the program code. In order to avoid this, we need to analyse the side effects of all programs. We know that side effects are due to communication (but *not* to single assignment), and combining this with tabulated information about the side-effecting behaviour of all the built-in functions allows us to perform a conservative side effects analysis, and only to wrap expressions in closures when we cannot guarantee that a side effect will not happen.

8.5 Atom analysis

Consider the following (legal) Erlang module

```
-module(foo[1]).

start() ->
  Pid = spawn(foo[1],foo[2],[foo[4]]),
  register(foo[3],Pid) ...

foo[2](X) ->
  foo[3] ! foo[4],
  ...
```

The atom `foo` is used here as a module name (superscript 1), a function name (2), a process name (3) and simply as an atom (4).

So what happens when we want to rename the module `foo`, for instance? We will have to analyse for each occurrence of an atom to which of the four categories it belongs.

Some uses can be read from the syntax, such as the module and function declarations. Other uses can be deduced from the atom being an argument to particular functions `spawn`, which takes a module, function and argument list as arguments, and `register`, which takes a process name and a process id. Other uses need to be traced through analysis of variable use, which we discuss next.

8.6 Process structure

An Erlang process is created by spawning it, an action that runs the function forming the body of the process and creates a *process id* or Pid. The Pid is needed to send messages to the process, and so the Pid will be passed around the program to facilitate access. In general, the Pid is the only way of referring to the process, but a process may also given a name through calling the `register` function.

Suppose that we want to refactor a program and name a particular process: this requires that we identify all the places in which the Pid is used. Take the concrete example of the process spawned in the `start()` function here:

```
-module(foo).

start() ->
  Pid = spawn(foo,foo,[foo]),
  foo(Pid).

foo(Pid) ->
  ... Pid ...,
  bar(Pid),
  ....
```

Data about the Pid flow into the `Pid` variable in `foo`, it is used in the body of `foo` and also passed to `bar`. We use a data-flow analysis to track all uses of the Pid, and in places where we cannot be sure about use or we do not warn the user that they need to check these variable uses themselves. As we said in Section 7.4, this is preferable to not performing the transformation at all.

8.7 Macros

Erlang has an integrated macro pre-processor, `epp`, and Erlang macros are in relatively common use. For example, the testing frameworks EUnit (Carlsson & Rémond, 2006) and Quviq QuickCheck (Claessen & Hughes, 2000) both use macros. Because we refactor source code, it is not sufficient to work with pre-processed code, and so when it is necessary for our analysis we use a combination of processed and non-processed codes, the latter being provided by the `epp_dodger` module. As an example, we need to use this to infer the precise binding information of variables used within a macro invocation.

It might be thought that Haskell was in a better position here, but paradoxically it is not the case. Haskell does not have an integrated macro mechanism, yet macros are needed for conditional compilation (e.g. of tracing) and for localisation of different architectures. The C pre-processor is therefore used instead, and this is substantially less tractable, not only because of its generality and power but also because there is no support in the Haskell front end for unexpanded macro calls (as there is for Erlang).

8.8 Conventions, frameworks and callbacks

Naming conventions are used to make programs easier to understand, but also they can have semantic significance in the context of Erlang libraries and frameworks; we give two examples here.

Within EUnit, any function with a name ending `_test` will be a single test, while one ending with `_test_` is a test representation. Changing these suffices will change the behaviour of the test suite. It is also assumed that tests for functions in the module `foo` will be contained in the module `foo_tests`. Finally, EUnit contains a number of predefined

macros which should not be modified. If refactorings are not to break test code, then these conventions need to be observed, and `foo_tests` should be renamed `bar_tests` when `foo` is renamed `bar`, for example.

The OTP is a set of libraries embodying generic behaviours of various kinds. In order to instantiate one of these behaviours, it is necessary to implement a set of callback functions: in this example of a generic server they are shown in *slanted* font:

```
init(FreqList) ->
  Freqs = {FreqList, []},
  {ok, Freqs}.
terminate(_,_) -> ok.
handle_cast(stop, Freqs) ->
  {stop, normal, Freqs}.
```

Any refactoring should not change the names or the arity of these functions, and so, for example, this must be taken into account in implementing the renaming and generalisation refactorings.

9 Testing and refactoring

The discussion of refactoring in this paper has so far been confined to refactoring *program code*, but programs or systems exist in a *wider infrastructure*: typically a program is built and deployed using some configuration scripts and make files (or equivalent), and is tested using some sort of test framework. As the program is refactored, and functions or modules change their names, for example, these changes need to be reflected in the wider infrastructure. We have investigated one particular aspect of this: how refactoring and testing interact for Erlang programs, and this is the main topic of this section. We also make some remarks about testing refactoring tools themselves in Section 9.4.

The most commonly used testing tools for Erlang include the OTP Test Server (TestServer) and Common Test (Common Test), EUnit (EUnit) and QuickCheck (Quviq). A common aspect of these frameworks is that the testing code is itself Erlang program text, albeit code that is idiomatic in its use of particular naming conventions, callback functions, meta-programming and the like.

Refactoring and testing interact in two distinct and complementary ways:

Extending existing refactorings. An existing refactoring, like renaming, will apply to test code as well as to the code under test; the refactoring should be extended to deal with this, in the context of the naming convention, for instance.

Refactorings of the tests themselves. Apart from those general refactorings, most testing frameworks also suggest a set of testing-framework-specific refactorings.

In early releases of Wrangler, testing frameworks were not taken into account. Therefore, when a program containing test code was under refactoring, things could easily go wrong without even a warning. For example, in EUnit functions with arity zero and names ending in `_test_` represent test generator functions; carelessly renaming a function whose name ends in `_test_` to one with a different suffix would break the test code.

Wrangler now takes into account the conventions of the common testing frameworks as well as supporting some refactorings that are specific to these testing frameworks. In the rest of this section we survey the idioms required by the testing frameworks, and then explain how Wrangler is extended to support consistent refactoring of test code when application code is refactored. We also briefly discuss testing-framework-specific refactorings.

Although we concentrate here on testing and on one specific programming language, it should be clear that the lessons apply equally well to other aspects of infrastructure, and to other programming languages, functional or not.

9.1 Testing tools for Erlang

In this section, we give a short overview of the commonly used systems supporting testing for Erlang, concentrating on the idioms that they require to achieve their effect.

9.1.1 Erlang/OTP test server and common test

In the Erlang/OTP Test Server and Common Test, a test suite is an Erlang module that contains test cases, and with a name of the form `*_SUITE.erl`; a collection of callback functions must be implemented in each such module. A test suite consists of a number of test cases, and test case is written as an Erlang function using a special coding pattern: each test case has generally three parts, describing the documentation, specification and execution of the test.

These parts are implemented as three clauses of the same function. The *documentation* clause matches the argument atom `doc` and returns a list of strings describing the test case. The *specification* clause matches the argument `suite` and returns the test specification for the test case. The *execution* clause implements the actual test case. It takes one argument, `Config`, which contains configuration information.

9.1.2 EUnit

EUnit is a lightweight unit testing framework for Erlang. Within EUnit the tester adds test functions or test-generating functions to a module as well as including the `eunit.hrl` header file. Test code can coexist with the application code in the same module, but it is also possible to put test code into a separate module. EUnit assumes that a module named `m_tests.erl` represents the test module for module `m.erl`.

A test function will have a name of the form `*_test`, and a test-generating function a name like `*_test_`. Symbolic representation of test data is used by test-generating functions to generate test objects. For example, the tuple {`generator, Module, Function`} is used to represent the test objects generated by calling the function `Module:Function()`, where `Module` and `Function` are module and function names respectively. A collection of predefined macros is provided to abbreviate the test code.

9.1.3 QuickCheck

QuickCheck (Quviq) is a property-based testing tool for Erlang. Programs are tested by writing properties (in Erlang syntax) and test case generators in the source code.

QuickCheck tests these properties with automatically generated test cases to see whether the system under test satisfies them. By default, any function with arity zero of the form `prop_*` is assumed to be a property.

When using QuickCheck to test, it is important to separate the testing of *pure* and *impure* functions. An impure function can modify the global state of the system, while a pure function will not. Impure operations are tested using an Abstract State Machine (ASM). ASM test cases are lists of symbolic commands, each of which binds a symbolic variable to the result of a symbolic function call.

For example, `{set,{var,1},{call,erlang,whereis,[a]}}` is a symbolic command to set the variable `1` to the result of the function call `erlang:whereis(a)`. The use of an ASM also requires the tester to implement a set of callback functions of the ASM.

9.2 Extension of existing refactorings

Since each of the three testing frameworks uses Erlang as the programming language for writing test code, much of the extension is achieved by existing functionality; the rest needs to address the particular idioms of the frameworks. This extension affects all the refactorings that change function and module interfaces.

In Wrangler the refactorings affected by this extension include *Rename function/module*, *Generalise function definition*, *Function extraction*, *Tuple function arguments* and *Move function definition to another module*. We now give a summary of the different aspects that needed to be addressed when extending these refactorings.

The testing framework(s) used. Checking which testing frameworks are used by the program under refactoring is trivial, since each testing framework requires a different header file to be included in the program.

Naming conventions. When a naming convention is enforced by a testing framework, the refactorer must ensure that this naming convention is observed. For example, when EUnit is used, renaming of a function ending in `test()` or `_test_()` to a name with a different ending (or *vice versa*) should generate a warning message; renaming the module `m` to some other name should also check whether there is a test module named `m_tests`, and if so, the test module should also be renamed.

Callback functions. Both Erlang/OTP Test Server and QuickCheck abstract state machines require the tester to implement certain callback functions. A callback function has a specified function interface that governs both the function name and the parameters accepted by the function. A refactorer should be aware of those callback functions, and always warn the user when the refactoring to be applied would turn a callback function into non-callback function (or *vice versa*).

Meta-programming. Each of the testing frameworks uses *meta-programming* to some extent. For example, symbolic function calls of the form

```
{call, ModuleName, FunctionName, Args}
```

are used by QuickCheck abstract state machines, and EUnit makes use of symbolic test data representation as mentioned earlier. Note that meta-programming is not restricted to testing frameworks, and the same format of symbolic function call as used by QuickCheck could also be used by normal application code in Erlang; however, inferring whether meta-programming is used by normal Erlang applications need deep data-flow analysis, and is not decidable in general.

Take `{call, ModuleName, FunctionName, Args}` as an example: Ideally a refactorer should be able to modify the module name, function name or the argument list whenever the module name or the function interface referred to is modified by a refactoring. However, given the fact that in Erlang atoms have multiple roles – syntactically module name, function name, process name are all atoms – and an atom could also act as a literal, it is also possible that the same tuple format is used to mean different things in different contexts.

Given this uncertainty, Wrangler takes a rather conservative approach. For refactorings like renaming, when Wrangler cannot infer whether an atom represents the module/function name to be named from syntactic information, it will check the context in which the atom is used. If the context indicates a high probability that the atom represents the module/function name to be renamed, it will rename it; otherwise it will leave it unchanged. In both the cases a warning message asking for the manual inspection from the user is given.

For refactorings that change a parameter of a function, Wrangler will try to keep the original function interface in the program, although its function body will be replaced with an application of the new function introduced. This is possible due to the fact that the Erlang allows the same function name to be redefined with a different arity.

Macros. Refactoring programs containing macros are generally supported by Wrangler, but early releases of Wrangler did not look into the actual definition of macros, and this turned to be a problem when refactoring QuickCheck code where macros are used very heavily. In fact, most of the QuickCheck library functions for writing properties are provided via macros; consider the example of the `FORALL` macro in

```
prop_gcd()->
  ?FORALL(X, nat(),
     ?FORALL(Y, nat(),
        ex:gcd(#rec{num1=X,num2=X*Y) == X)).
```

where it is used to represent universal quantification in a way that allows tests to be generated for the property.

The heavy use of macros and the complexity of macro definitions make it sometimes impossible to resolve the binding structure of variables without looking into the actual macro definitions, as the above example function shows. The study of QuickCheck has led us to improve the way in which macros are handled in Wrangler. Two kinds of ASTs are kept during the refactoring process, one with all macro applications expanded and another with macro expansion bypassed. The former is used to infer the accurate binding structure of variables, which is then passed on to the latter.

Coding patterns. The Erlang/OTP Test Server and Common Test framework ask testers to write test cases following a special coding pattern. For example, a test case generally takes one parameter, has three function clauses representing the documentation part, the specification part and the execution part, and each function clause takes a specific pattern to match. In this context the refactorer should make sure that this coding pattern is not violated during the refactoring process.

Take the *Generalisation* refactoring (as described in Section 2.1) as an example. This refactoring certainly changes the function interface, and generalisation of a test case function will make it no longer a test case. With Wrangler, this kind of violation is again avoided by keeping the original function in the program, but its function body will become an application instance of the new function.

9.3 Testing-framework-specific refactorings

To make Wrangler testing-framework-aware, we aim to make Wrangler not only be able to refactor application code and test code consistently but also be able to support testing-framework-specific refactorings. Our study of the three testing frameworks shows that different refactorings will supplement different testing frameworks.

9.3.1 Erlang/OTP test server and common test

Test code written under the Test Server and Common Test framework has a rather constrained top-level structure because of the coding pattern followed; however, our case studies show that most test cases have very similar structure, and the *copy*, *paste*, then *modify* style of editing is very heavily used, which results in substantial amount of duplicated code.

Tool support for duplicated code detection and elimination would help to provide better abstraction of some repeatedly used functionalities, improve the code structure, reduce the size of the code and make it much easier to understood. Together with our project partner from Ericsson, Sweden, we have used Wrangler's support for duplicated code detection and elimination (Li & Thompson, 2010b); the results are discussed in Li *et al.* (2009).

9.3.2 EUnit

EUnit code could also be helped by refactoring. The following refactorings could be added to Wrangler:

- Convert tests written in plain Erlang into EUnit tests.
- Group a set of EUnit tests into a single test generator.
- Move EUnit tests in an application module to a separate test module.
- Normalise EUnit tests to a standard pattern.
- Extract common setup and tear-down code into *fixtures*.

9.3.3 QuickCheck

With QuickCheck, our research has been focused on refactorings that create properties, and refactorings that change the structure of the existing properties. For example, using

the techniques of similar code detection and elimination discussed in the next section, it is possible to turn a set of common test cases into a number of calls to a single function. From these calls it is possible to extract a QuickCheck property by the following steps:

- Firstly, identify all the calls to a particular function, and extract their arguments into a list of tuples, each tuple representing one call to the function.
- The tests can then be turned into a property by choosing `oneof` the list of test data, with the tuple chosen becoming the arguments to the call of the test function. Here `oneof` is a simple example of a QuickCheck *generator*.

We have presented our experience of extending Wrangler to accommodate testing framework programming idioms, and discussed our ongoing work to support testing-framework-specific refactorings. While this research focuses on Erlang and Wrangler, the same problem should apply to other programming language domain and refactoring tools as well.

9.4 Testing refactoring tools

While building refactoring tools we have become aware that they can be particularly problematic to test. One aspect of this problem is that we need to test two quite different aspects of the systems. Firstly, we have to ensure that refactoring does indeed preserve the *semantics* of programs, but secondly, we also have to ensure that the *appearance* of the results is acceptable to the programmer. In practice we have tended to concentrate on the former, while feedback from users has helped us to substantially improve the layout of refactored code; in fact, we use QuickCheck in testing Wrangler (Li & Thompson, 2007).

More recently we have conducted an experiment using random testing of refactoring tools. Our approach is to generate random programs using a simple attribute grammar-based generator within QuickCheck. We then refactor with a random refactoring, generate program inputs randomly and test whether the old and new programs agree on these inputs. The experiment was successful to the extent that it identified bugs in Wrangler. More details are reported in Drienyovszky *et al.* (2010).

10 Clone detection

While it has been our aim to provide the fundamental refactoring support for Haskell and Erlang, users have asked for help in applying the refactorings. To do this, we have developed a variety of *decision support* tools that help users to identify and apply refactorings. These include tools to report on code clones within projects (which we discuss further in this section), in subsequent sections we describe tools to identify potential problems in the module structure of projects as well as detecting more localised 'bad smells' within modules. We concentrate here on describing the facilities in Wrangler; details of clone detection in HaRe can be found in Brown & Thompson (2010).

We characterise a software clone as two or more fragments of code – expressions or sequences of expressions – that share a non-trivial common generalisation. As shown in Figure 9, we report on all such clone sets, and for each clone show

- the common generalisation, in the form of a function definition, and

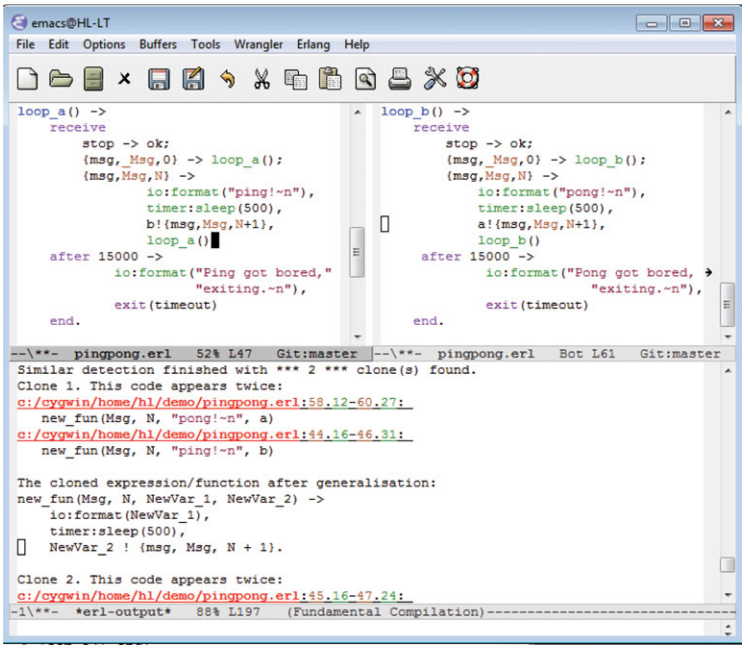

Fig. 9. (Colour online) Clone detection and elimination in Wrangler.

- the instances of the clone, including the parameters which embody how it instantiates the generalisation.

To take a concrete example for the code

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

```
loop_b() ->
  receive
    {msg, _Msg, 0} -> ok;
    {msg, Msg, N} ->
      io:format("pong!~n"),
      a ! {msg, Msg, N+1},
      loop_b()
  end.
```

we report the clone identified above by *slanted code* as this function

```
new_fun(Msg,N,New_Var1,New_Var2) ->
  io:format(New_Var1),
  New_Var2 ! {msg, Msg, N-1}.
```

and the two calls giving the two clones like this

```
new_fun(Msg,N,"ping!~n",b)
```

```
new_fun(Msg,N,"pong!~n",a)
```

In a project of any size – and we are able to report on clones within projects containing hundreds of KLOC – there will potentially be large numbers of clones to report, and so we have to specify threshold parameters for reporting. These include the size of the clones (in number of expressions and number of tokens), and the size of the clone class, i.e. the

numbers of duplicates. We can also measure how much generalisation has gone on by comparing the size of the `new_fun` with its instances (this ratio we call the *similarity*) and also by the number of `New_Vars` introduced by the generalisation, each of which represents a position in which the clone instances differ.

Search for clones can take two forms: It is possible to produce a report which identifies *all* clones in a project, or it is possible to identify all the clones of a *particular code fragment*. The former report allows clone exploration, the latter is a more fine-tuned approach to eliminating precisely the clone that makes sense within a particular context.

Our reports give *decision support* data to the user, and on the basis of this the user can decide which clones to eliminate, and in which order. The user will also have to give meaningful names to the `new_fun` and `New_Vars`, and so this cannot be a 'push button' operation, but instead needs the insight of a programmer to apply clone elimination.

Clone identification is made both fast and accurate through a hybrid clone detection mechanism. Clone *candidates* are identified by means of a string matching algorithm operating over strings generalised from the program code. The candidates are then checked using an AST matching, which supports analysis of static semantics of the program. While the AST algorithm would be too slow to be practical on its own, it can be used efficiently on the candidates only.

Incremental clone detection is supported by caching results and identifying where programs have changed since the previous check; this allows clone detection to be performed regularly – e.g. as part of a nightly build process – without incurring too heavy a burden; it also makes interactive clone detection more efficient in larger projects. Further details of the implementation, including how it is implemented efficiently and incrementally, can be found in Li & Thompson (2009a, 2010b, 2011b).

10.1 Case study: clone detection

.

The clone detection tool was piloted in a case study undertaken at Ericsson AB, together with the engineers Adam Lindberg and Andreas Schumacher. The purpose of the case study, which examined a file of some dozen tests written in Erlang, was to allow the engineers to understand the test code and to make it more maintainable. The existing tests were typically straight-line code, with some use of simple macros; clone detection allowed the engineers to identify common code patterns, which were candidates for clone elimination through introducing appropriate functions.

The clone elimination process went through 12 steps; the sizes of the file versions are given in Figure 10. The initial report shows 42 clones, with the most common one being cloned 16 times; a summary of this report is in Figure 11, and more details in Li *et al.* (2009). The initial report uses the default threshold parameters, namely minimum length of expression sequence: 5; minimum tokens: 40; minimum appearances: 2; maximum new parameters: 4 and minimum similarity: 0.8. The case study was terminated after 12 iterations, but clone detection on version 13 still shows a variety of clones, and it would be possible to further reduce the size of the file by hundreds of lines.

A key lesson coming out of the case study is that it is necessary to work with domain experts when eliminating clones for the following reasons.

Version	LOC	Version	LOC	Version	LOC
1	2658	6	2218	10	2149
2	2342	7	2203	11	2131
3	2231	8	2201	12	2097
4	2217	9	2183	13	2042
5	2216				

Fig. 10. File sizes in the clone detection and elimination case study.

	Size (LOC)	Occurrences	Total parameters	New parameters
Median	17	2	4.5	2
Mean	19.1	3.4	4.8	2.3
Maximum	89	16	11	4
Minimum	7	2	0	0
Largest clone	89	2	2	2
Second largest	61	3	3	3
Most occurring clone	7	16	0	0
Second most occurring	9	14	1	1
Most parameterised	21	2	11	4
Number of clones	42			

Fig. 11. Initial clone data for the case study, using default threshold parameters.

- A clone as identified by the tool may not be meaningful within the application domain. It might, for example, be made up of two sub-clones, each of which is meaningful. Also, it might be that some code is 'accidentally' attached to the beginning or end of the actual clone, simply because of the context in which the clone appears. In both of these cases, which occurred in the case study, the engineer needs to identify the specific code to be extracted into a function.
- Once a clone is identified and becomes a function, it needs to be named; moreover, the parameters of the new function also need to be named. This can only be done by someone with domain knowledge.
- It is also a matter of choice as to how much to generalise. Is it meaningful to introduce a function with four new parameters, or would two similar functions with two parameters each be easier to name and understand? There is also a question of whether it is better to introduce a number of layers of generalisation, one parameter at a time or a single generalisation with N parameters. In both the cases it is an engineering decision.

Other conclusions from the case study were as follows:

- As a general principle we found it more useful to replace clones bottom up, rather than top down; for instance, in the first step we chose to eliminate the most commonly occurring clone rather then the largest one. One reason for this is that it is easier to understand the operation of a smaller code fragment, and so to name it

appropriately. Once smaller clones are replaced by appropriately named function calls it becomes easier to name larger clones.

- It was important to provide users with the two modes of clone detection: initial exploration identified some clones, but examining these led to others being eliminated. That was because the identification mechanism might have included some spurious code, or because a clone has not been listed due to the choice of threshold parameters; in either case it is necessary to search for clones of a specified code fragment before performing elimination.
- As with general refactoring, it is usually the case that clone eliminations are interspersed with manual refactorings that enable, for example, a larger clone to be identified.

Finally, we observe that version 13 is 23% smaller than version 1, and further reductions are clearly possible. The result is not just a more compact representation of the test code but also better structured code with which it will be easier to construct new tests.

10.2 Case study: incremental clone detection

In a long-running project, it would be typical to perform a set of checks on a regular basis, and one of these could be clone detection. For a large project, the cost of detection can be substantial, and so an *incremental* approach is desirable. We report on this in detail in Li & Thompson (2011b), where we show that in a number of large projects we can cut clone detection time periods by up to 85%. Figure 12 shows the time taken to detect clones in three ongoing projects: Wrangler itself, and test codes for a telecoms application and Erlang itself. The reduced time periods for later versions come from applying incremental detection leveraging results for earlier versions.

11 Module bad smells

One of the fundamental refactorings we provide in Wrangler is to move a group of functions from one module to another. If the target module does not exist already, then it is created by this move. Such a refactoring allows us to change the module dependency structure of a system in an incremental way. We believe that this is the right way to go about restructuring a system, rather than trying to achieve a one off wholesale change which would be difficult to document or comprehend.

The question is, then, which functions to move? We provide information of different kinds.

- Reports on cycles in module dependencies; these are of two kinds:
  - — *Intra-layer dependencies*: Here a cycle in the module graph reflects a strongly connected component which spans more than one module. These can be ameliorated by grouping the SCC in a single module, or will simply be accepted as they stand.
  - — *Inter-layer dependencies*: Here a cycle does not reflect mutual recursion, but instead can stem from a crossing of layers in a layered architecture. A typical example is a function which is a client of a library which by its utility is itself

	KLOC	Files		Incremental		Standalone	
		Changed	Total	Time	Clones	Time	Clones
Wrangler							
0.8.7	42.5	70	70	15	18	15	18
0.8.8	44.2	59	80	10	21	15	21
0.8.9	47.4	44	83	8	26	16	26
0.9.0	48.0	9	84	2	26	16	26
0.9.1	48.1	3	84	3	26	17	26
Test Suite							
V0	24.0	26	26	560	371	560	371
V1	23.9	3	26	90	361	550	361
V2	23.9	1	26	54	357	550	357
V3	23.8	2	26	90	346	550	346
V4	23.7	2	26	80	338	540	338
Erlang							
R13B-03	244.3	306	306	94	78	94	78
R13B-04	245.5	71	311	36	79	97	79
R14A	250.8	108	327	40	82	95	82
R14B	251.9	39	321	28	81	94	81

Fig. 12. Incremental versus Standalone clone detection.

used by other library functions: the solution is to move this from the client to the library.

- Improper inter-module dependencies, under which a function whose dependency analysis suggests that it should not be exported is nevertheless exported.
- Partitioning of the set of functions exported by a module into clusters of functions which share an affinity.

We applied this to our own system, Wrangler-0.8.7, made up of 56 modules, with 40k lines of code. The structure of the Wrangler code is that each refactoring is implemented in a separate module, with shared functionality contained in library modules. We found a number of improper dependencies where distinct refactorings were directly sharing implementation code (that was not in a library).

We also identified some inter-layer cyclic dependencies which were solved by splitting up modules. Finally, we found that the `refac_syntax_lib` library contained seven clusters, suggesting that we should split it up.

As was the case in Section 10, we are not suggesting that any of these transformations could be done automatically: for example, the library was split into fewer than seven clusters, each representing a different functionality, rather than splitting it as the report might have suggested.

Details of the implementation of the dependency checks and further examples are contained in Li & Thompson (2010a).

11.1 Other 'bad smells'

We also report on local properties of modules such as the depth of nesting of `receive` or `case` statements; instances of variables (definition and use), non-tail-recursive functions used as process bodies and so forth.

This mechanism is used in the Erlang Solutions e-learning system (Erlang E-learning, 2011) to provide feedback on the style of solutions submitted for online assessments. For example, based on the kinds of pattern used in a pattern-matching definition, Wrangler can suggest alternative ways that the student might solve the problem in question.

The extensions described in Sections 12.1 and 12.2 provide the user with the mechanism to describe 'bad smells' for themselves, and this is of particular value in a pedagogical application like this, where a user might wish to tune automated feedback question by question.

12 Extensibility

As Wrangler was initially developed – and throughout the development of HaRe – someone wanting to add a new refactoring to the system would need to learn a whole collection of internal APIs as well as the concrete representation of Erlang or Haskell within the tool. This had the practical consequence that new refactorings were only added by the developers.

Recent releases of Wrangler have included an API designed to make implementations of new refactorings substantially easier as well as a domain specific language (DSL) for describing complex composite refactorings. We describe these in more detail in this section, and conclude by presenting a case study of their use.

The work described in this section gives extension points for Wrangler accessed by writing Erlang programs using the API and DSL; by contrast, our work in embedding Wrangler in Eclipse (ErlIDE) (ErlIDE, n.d.) is supported by the Eclipse plug-in framework that enables extensions to Eclipse to be added, written in Java; in our case that Java code happens to call out to a separate Erlang refactoring process, but plays no part in the implementation of that process, which is pure Erlang.

12.1 An API for describing refactorings

Wrangler contains an API to describe new refactorings (Li & Thompson, 2011a). To make the API easier to use for an experienced Erlang programmer, we hide the details of syntactic and semantic representations, and instead allow the syntax to be represented by *Erlang concrete syntax*, augmented with meta-variables that are Erlang variables ending with the '`@`' symbol. Using these, which we call *code templates*, we are then able to describe transformations by rules giving the 'old' and 'new' codes, together with preconditions for the refactoring and a description of how the rules are to be applied across the syntax tree. These are illustrated in more detail in the case study in Section 12.3.

We specify what it is to be a refactoring by defining an OTP *behaviour* that encapsulates the generic part of a refactoring process so that the user only has to implement the call-back functions that embody the functionality that is particular to the refactoring under consideration. This means that the new refactorings are 'first-class citizens' in that users are able to invoke the newly defined refactorings within Emacs and ErlIDE (ErlIDE, n.d.), and so preview and undo changes that their transformations make, just as for the refactorings built into Wrangler.

In contrast, the API available in HaRe is of lower level, and requires users to understand and manipulate the syntax representation of Haskell within Programatica, and the semantic information contained therein.

12.2 A domain-specific language for refactoring

In their recent study on how programmers refactor in practice, Murphy-Hill *et al.* (2009) point out that 'refactoring has been embraced by a large community of users, many of whom include refactoring as a constant companion to the development process'. However, following the observation that about 40% of refactorings performed using a tool occur in *batches*, they also claim that existing tools could be improved to support batching refactorings together.

Complementing the API, we have also built an embedded, domain-specific language in Wrangler (Li & Thompson, 2012a) for describing composite refactorings: refactorings that are composed from a number of primitive refactorings. The DSL is implemented using Erlang macros that make DSL programs straightforward to read and write. The DSL gives a powerful and easy-to-use framework that allow users to script their own reusable composite refactorings to carry out large-scale *batch* refactorings in an efficient way. As for the API, these scripts can be run through the Emacs interface, and so the results can be previewed and undone. (Doing this in ErlIDE is more problematic due to the workflow required for refactorings; it remains a future work.)

This facility is a *language* rather than simply an API as it allows control of a number of different dimensions of the refactoring.

- Refactorings in the core Wrangler implementation take arguments to describe what is to be done. For example, this specific function is to be renamed in this way. The DSL allows for refactorings such as renaming to be paramterised instead by conditions that identify the functions to be renamed, and functions that supply the renaming itself. These more abstract descriptions *generate* collections of concrete refactorings for Wrangler to execute.
- It is possible to control the *granularity* of composite refactorings, and this is essential for preserving correctness in many cases. For example, suppose that a sequence of renamings depends on each other, since we might want to generalise the functions `foo/3`, `foo/2` and `foo/1` in that order; if the first fails, the rest should fail too, and we make this a *transaction*. On the other hand, if we want to make a batch transformation of names from '`camel_case`' to '`camelCase`' across a project, the refactoring will still be successful (pin not breaking the semantics of the program) even if one of the component renamings fails: we therefore do not make this a transaction.

- We can also choose to make refactorings described by the DSL *interactive*, that is taking user input during their execution, and also *iterative*, that is applied repeatedly under user control. An example of this would be in removing instances of code clones, where it should be possible to query the user at each instance of the clone whether or not it should be removed.

Examples of use of the DSL are discussed in more detail in Li & Thompson (2012c) as well as in the next section.

12.3 Case study: removal of error macros

As a case study of the API and DSL together, we have worked with Quviq AB to develop a refactoring to assist their testing work. Quviq tests systems using the QuickCheck tool for property-based testing. In testing state-based systems, a state machine model is written in Erlang, and the actual system (which may be written in another language like C) is taken through randomly generated test cases for the model. Once a fault in the system has been discovered, it is not useful to keep identifying the same fault, and so testers modify the model by adding calls to macros, such as `?system_bug_007`, to bypass the erroneous code, as in the statement

```
case ?system_bug_007 of
    true  -> ... known_failure ...;
    false -> ... usual_behaviour ...
end
```

On shipping the model to the customer, this fault avoidance code needs to be removed, and so, for example, the `case` statement above needs to be replaced by the `usual_behaviour`. This is not simply a matter of replacing `?system_bug_007` by `false`: the resulting code can often be simplified substantially afterwards, and the automated refactoring was required to do that. Prior to automation this was a tedious exercise for the tester: for example, one of the files that we examined contained 43 uses of these macros, in a number of different contexts. In the most complex of these our refactoring reduces a 14 line nested `case` statement containing two macro calls to five lines of code.

We use the API and DSL together to achieve the automation. The simplest way to achieve the required simplification is to replace macro invocations like `?system_bug_007` by `false`, but this still leaves the code cluttered. In the earlier example, the case statement needs to be collapsed from

```
case false of
    true  -> ... known_failure ...;
    false -> ... usual_behaviour ...
end
```

to the `usual_behaviour` by a simple case of symbolic execution. Other tidying up is also possible, e.g. by removing concatenation with an empty list. This is all implemented in the refactoring `refac_bug_cond` shown in Figure 13.

- At the top level we implement the callback function `transform` that specifies the transformation to be done: this specifies that a list of rules is applied to the full

```
transform(_Args=#args{search_paths=SearchPaths})->
    ?FULL_BU_TP([replace_bug_cond_macro_rule(),
                 logic_rule_1(),
                  .....
                 if_rule_1()], SearchPaths).

replace_bug_cond_macro_rule() ->
    ?RULE(?T("Expr@"), ?TO_AST("false"),
          is_bug_cond_macro(Expr@)).

logic_rule_1() ->
    ?RULE(?T("not false"),?TO_AST("true"),true).

if_rule() ->
    ?RULE(?T("if Conds1@@, false,Conds2@@ -> Body1@@;
                 true -> Body2@@
              end"), Body2@@, true).

is_bug_cond_macro(Expr) ->
    api_refac:type(Expr) == macro andalso
        is_bug_cond_name(?PP(Expr)).

is_bug_cond_name::string()-> boolean().
is_bug_cond_name(Str) -> ..check the macro name....
```

Fig. 13. Eliminating bug preconditions: key aspects of `refac_bug_cond.erl`.

program tree (`FULL`) in a bottom-up way (`BU`), emitting another program tree (`TP`, for 'type preserving'). Note that here we use the terminology of strategic programming (Lämmel & Visser, 2003; Bravenboer *et al.*, 2008).

- At each node of the tree we attempt to apply the rules in the list in turn, until one is applied successfully.
- The principal rule is `replace_bug_cond_macro_rule` which replaces any expression that is a *bug condition macro* by the literal `false`.
- Rules are signalled by the three argument macro `RULE`: the first argument is the code to be replaced, the second is its replacement and the final argument is the precondition for the rule to be applied. Note in the example that we use meta-variables ending with '`@`' in the templates matching the old code, and have access to these in the new code and precondition.
- The most complicated rule shown here is the rule for simplifying `if` statements, `if_rule`. Here the variables ending `@@` will match *sequences* of expressions so that the pattern `Conds1@@, false,Conds2@@` will match any sequence of expressions in which one of the expressions is the literal `false` (note that a comma-separated sequence of conditions in Erlang is interpreted as the conjunction of the values).
- In a general refactoring there are also callbacks to handle interactivity (how the user is prompted etc.), selection of focus in the editor and a precondition for the refactoring itself. In this case these all have the default definition.

```
composite_refac(_Args=#args{current_file_name=File,
                            search_paths=SearchPaths}) ->
    Refacs=?refac_(inline_var, [File,
                                fun({_F,_A}) -> true end,
                                fun(MatchExpr) ->
                                        ?MATCH(?T("Var@=Expr@"),MatchExpr),
                                        api_refac:type(Expr@)==variable
                                end, SearchPaths]),
    ?non_atomic(Refacs).
```

Fig. 14. Using the DSL to give a generator for inlining all redundant variables.

- In total, the rule list contains 14 rules, each implemented in a few lines of code, and each being clearly semantics-preserving. Together with the callback functions and auxiliary definitions, the file consists of some 160 lines.

The refactoring presented in Figure 13 will drastically simplify code, but can still leave further work to be done, as in the case of

```
route_data_next(S,_, [{{SrcKind,SrcId}, Dst, Val}], _) ->
    NewS = set_gateway_pending(S, SrcKind, SrcId, false),
    S2 = NewS,
    copy_to_destination(S2, Dst, Val).
```

where the simplification has resulted in code with a redundant variable introduction (`S2`). In this case we want to inline that definition, replacing all instances of `S2` by `NewS`. This can be described by a composite refactoring in the DSL, and specifically by means of a *generator*, as shown in Figure 14.

The function `composite_refac` returns a generator, which is signalled by the assignment

```
Refacs = ?refac_(...)
```

The macro `?refac_` generates instances of `inline_var` according to the functions in the second argument. The first argument gives the file in which the transformations take place, the second describes which functions it should be applied to (in this case *all* of them) and the third gives the criterion for application: an assignment of the form `Var@=Expr@` in which the `Expr@` is itself a variable. The result of the function, `?non_atomic(Refacs)` means that the refactoring is not treated as a transaction: the failure of any of the particular refactorings does not lead to the failure of the whole.

We now have two refactorings, and these two can be put together into a single one using the DSL. This final composite gives a 'push button' solution to the bug precondition problem, and thus simplifying the work of Quviq test engineers.

Another use of the API and DSL is to assist in the 'API upgrade' problem: when an API evolves, it is necessary to upgrade code using it. The simplest way of doing this is to provide an adapter module that implements the old API in terms of the new: using the DSL and API we have built a system that transforms the adapter module into a set of refactorings that transforms the client code to work with the new API; this is reported in detail in Li & Thompson (2012b).

13 Related work

Refactoring. The term 'refactoring' was first coined in the early 1990s in two PhD theses (Griswold, 1991; Opdyke, 1992), and was brought to wider prominence by Fowler's (1999) eponymous book; a broad survey of the first decade of the development of refactoring and refactoring tools is provided by the survey (Mens & Tourwé, 2004). A series of workshops on refactoring tools, including WRT'12 (Sommerlad, 2012), have brought together researchers and engineers building tools to refactor languages, and in papers and discussions there have been developments and observations complementing ours here.

Refactoring has received a lot of attention in the OO programming community, with tools produced for Java, not only in Eclipse (Gallardo, n.d.) but also in other IDEs (NetBeans, n.d.; Lahoda *et al.*, 2012); C# (ReSharper, n.d.) and Smalltalk (Roberts *et al.*, 1997) among others. The applicability and use of these and other tools are compared in Katić & Fertalj (2009) and the general uptake and application of refactoring ideas and tools are evaluated in Ge *et al.* (2012) and Vakilian *et al.* (2012).

Program transformation for functional programs. Despite the fact that one of the founding works of the area (Griswold & Notkin, 1993) addressed LISP, there has been relatively little work on refactoring functional programs; there is, however, a substantial heritage of program transformation for functional programs, beginning with the seminal work of Burstall & Darlington (1975) surveyed in Darlington (1982). More recently, Lämmel (2000) has used Strafunski to describe generic and functional refactorings; this work has tended to focus on principles of program transformation rather than on tooling for specific, complete languages. Strafunski (Lämmel & Visser, 2003) is a tool for strongly typed transformation written in Haskell, and is itself inspired by Stratego (Bravenboer *et al.*, 2008), which is untyped.

What the majority of the functional transformation tools have in common is the fact that they are not designed to work with concrete syntax, and so it is difficult, if not impossible, for the tools to preserve layout and comments. Others, such as PATH (Tullsen, 2002), support Haskell program transformation for program derivation or for optimisation: in both the cases these improve systems *locally*, whereas the refactorings that we target are large-scale, not least because these are the transformations most in need of automated support.

Refactoring functional programs. Turning to work that does address refactoring functional programs, the team at Eötvös Loránd University, Budapest, have built the RefactorErl tool (Horváth *et al.*, 2008) for Erlang refactoring. Their approach differs from ours in a number of ways:

- Their tool has a dedicated lexical analyser and parser, in contrast to our reuse of the facilities of the `syntax_tools` library. This has the advantage of allowing the team complete control of syntax analyses, but requires maintenance to follow any changes to the existing distribution.
- Their system stores the program information in a database rather than our choice of using syntax trees with annotations for semantic information. This has the advantage of supporting a series of static analyses (Tóth & Bozó, 2012), but the disadvantage of

making the tool more heavyweight: startup time can be longer than the corresponding time for Wrangler.

- We have added a number of higher-level decision analysis tools to our system, including clone detection and module 'bad smell' identification, and linked these with the appropriate refactorings to support the correction of identified faults.
- The RefactorErl team has developed tools to analyse and remodularise (Lövei *et al.*, 2008; Lövei, 2009) systems. These have tended to automate under user guidance. Our systems instead provide users with reports on the basis of which users decide to make certain changes; we do this because, in practice, it is often impossible automatically to determine the appropriate remodularisation, say, fully automatically without any user input.
- In contrast with our approach of allowing users to use Erlang concrete syntax in writing new refactorings (Li & Thompson, 2012c), the RefactorErl team's support for user access to the syntax (Kitlei *et al.*, 2008) requires users to understand the details of the internal representation to describe syntactic fragments.

The projects can be seen as complementing each other. Our tool has been designed for interactive work, while the higher overhead of populating the database in RefactorErl could potentially pay off when working with larger code bases. More recent developments to RefactorErl, as reported in Tóth & Bozó (2012), concentrate on its facilities for analysis rather than transformation.

Tidier (Sagonas & Avgerinos, 2009) is another refactoring tool for Erlang; however, it differs from Wrangler and RefactorErl both in the kind of refactorings supported and the design philosophy. Firstly, the refactorings in Tidier are more limited in scope (currently, they are mostly clause-local); secondly, Tidier is designed to be run fully automatically and requires no interaction from user; since layout is not preserved, the tool is better suited for use during the 'make' process, rather than to produce source code recognisable to the original author. HLint (Mitchell, 2011) is a similar tool for Haskell that also identifies 'local', expression-level refactorings; HLint is highly user-configurable, but at the time of writing HLint requires users to perform transformations for themselves.

Papers of ours on more specific topics – such as clone detection and elimination – have more detailed literature reviews.

Testing and refactoring. There is little other work on automated testing of refactoring tools. The most prominent example is Daniel *et al.* (2007), a similar study to ours, but done for refactoring engines integrated in IDEs for mainstream languages. This also uses program generation, but the program generator described is specific to the language they use and it can be parametrized by code fragments, so it would be difficult to adapt to other domains, rather than our more general approach. They test the results of refactorings in a different way too: they test handwritten structural properties as opposed to behavioural equivalence, and do this by testing the results of two different refactoring engines against each other, rather than testing the old and new code directly.

Building on this work, Soares (2012) provides a system whose main contributions are 'its technique for generating input programs and its test oracles for checking behavioral preservation based on dynamic analysis'. Our earlier work in Li & Thompson (2007)

describes using QuickCheck for testing Wrangler. This did not use random program generation, refactoring a static code-base instead, but generating random refactorings of that code-base; we then check successful compilation as a property of all refactorings, as was the case for Daniel *et al*. (2007) as well as other properties of individual refactorings.

While there is an established literature on refactoring test code written in the 'xUnit' style (Meszaros, 2007), we are not aware of xUnit-awareness having been implemented in any existing refactoring engines. The analysis we perform requires knowledge of the Erlang pre-processor, and similar work to this is reported in Kitlei *et al.* (2010).

Clone detection. We have introduced a mechanism which combines the speed of string matching with the accuracy of AST-based detection; for larger projects it is important to check the clones added by changes rather than rechecking the whole project: a method for this is outlined in Li & Thompson (2011b).

A survey by Roy *et al.* (2009) provides a qualitative comparison and evaluation of the current state of the art in clone detection techniques and tools. Overall, there are text-based approaches (Baker, 1992), token-based approaches (Baker, 1995; Li *et al.*, 2006), AST-based approaches (Jiang *et al.*, 2007; Bulychev & Minea, 2008; Evans *et al.*, 2008;) and program dependency graph-based approaches (Komondoor & Horwitz, 2001). AST-based approaches in general are more accurate, and could report more clones than text-based and/or token-based approaches, but various techniques are needed to make them scalable.

Closely related work to ours is by Bulychev & Minea (2008), who also use the notion of anti-unification to perform clone detection in ASTs. Our approach is different from theirs in three ways: firstly, our hybrid approach is more efficient and scalable, but still accurate; secondly, we report clone classes, while they only identify clone pairs; finally, their language-independent approach means that they cannot reflect the particular static semantics of a particular language like Erlang.

In Brown & Thompson (2010) an AST-based clone detection and elimination tool for Haskell programs is described; their approach works on small Haskell programs, but scalability is a problem that the authors do not address.

ClemanX is an incremental tree-based clone detection tool developed by Nguyen *et al*. (2009). Their approach measures the similarity between code fragments based on the characteristic vectors of structural features, and solves the task of incrementally detecting similar code as an incremental distance-based clustering problem. Göde & Koschke (2009) describe a token-based incremental clone detection technique, which makes use of the technique of generalised suffix trees.

Extension. We have introduced a template-based API for describing new refactorings and a domain-specific language for scripting refactorings. Rather than introducing a new language, these are integrated with Wrangler and use Erlang concrete syntax and data structures, which will be familiar to Wrangler users.

Others have made progress in this area, including Leitão (2002), who describes a 'pattern language' in LISP syntax designed to assist the description of transformations, and Kitlei *et al*. (2008), which supports the description of nodes of the AST by means of an Erlang-style syntax. While each leverages the support of the host language, neither supports concrete syntax description of fragments, nor do they directly support the collection and

collation of context data needed to describe refactoring preconditions as we do in Li & Thompson (2011a).

Recent versions of NetBeans include a facility to define custom 'declarative' refactorings (Lahoda *et al.*, 2012), and these can be described by concrete syntax templates; what the system currently lacks is the ability to make new refactorings interactive and selective (as well as transactional), as provided by our DSL (Li & Thompson, 2012a). Another approach to specifying refactorings in a high-level way is given by Schaefer & de Moor (2010), where simpler micro-refactorings are put together subject to constraints to define whole refactorings. In some ways this approach resembles our DSL, but it also allows description of refactorings from scratch: it would be interesting to see how this approach adapts to a functional language. Hills *et al*. (2012) give an example of 'scripting' refactorings in the Rascal system in a way similar to our DSL approach.

14 Reflection

Reflecting on the experience of building HaRe and Wrangler leads us to draw some conclusions. We begin by noting that implementing refactoring tools for particular programming languages gives us a perspective on the languages themselves.

Haskell. In managing imports and exports it is useful to be able to hide particular bindings, and indeed Haskell allows us to hide bindings on import. Unfortunately, the dual operation of hiding a binding on export is not in the language, and so explicitly hiding one of a large number of exports is impossible. Instead, it is necessary to export all but that one binding, which is neither readable nor easily maintainable.

We showed the example of turning a `data` type into an abstract data type, and that is supported by Haskell. As a part of that we use field names as selectors, and it would be useful for those field names to be a part of definition of the standard types such as lists, since this facilitates the transformation. Without this we need to be able to recognise that `head` and `tail` are indeed the functions that we need.

There are also complexities introduced by two of the more esoteric (and controversial) aspects of the Haskell-type system: The monomorphism restriction and defaulting. The former puts restrictions on the application of eta-reduction and expansion, and combining both can result in the meaning of a program changing when an unused definition is removed after unfolding, for example.

Layout of Haskell programs is significant, but also idiosyncratic: Program layouts of different programmers can be very different, and so ensuring that the layout is preserved as much as possible is very important for the output to be acceptable to the author of the program being refactored. The Haskell standard does specify that a tab stop should be interpreted as eight characters, removing one cause of problems for other languages.

Erlang. The Erlang tradition has been not to give types to programs, but this has changed. Previously there were two notations for types, `-spec` as used by the Dialyzer tool (Sagonas, 2007), and `@spec` as used in the Edoc documentation system; the latest releases of R14 have unified this notation to `-spec`. We expect that this will provide a tipping point after which we can expect that stating types will become the norm, and this will significantly aid

tool writers. For a refactoring tool, in cases such as that in Section 8.2 it should be much clearer from the type declared for a function whether or not a catch-all is needed; indeed, use of a type synonym in a context like that will signal the scope of a wider transformation of types within a whole set of functions. Type information will also aid implementers of other tools, such as QuickCheck, which generates random data of particular types.

Dynamic languages make the tool writer's job more difficult, and this is spelled out in detail for refactoring Erlang in Section 8. Given that this is central to the ethos of the language, we do not expect to see changes to Erlang in this respect. Neither do we see any imminent changes to the way in which the same atoms can be used for multiple roles in Erlang: enforcing some sort of lexical separation between these different categories would be too disruptive a change.

Do it yourself? Our aim in writing HaRe and Wrangler was to reuse other frameworks as much as possible. This led us to choose Programatica and Strafunski for HaRe, and `syntax_tools` for Erlang. The experience of HaRe was that this got us started quickly when we began the project in 2002, but both frameworks were not supported soon after that point. An alternative choice were we to begin again now would be to use GHC itself, and in particular to exploit the API for the GHC front end (GHC API, n.d.) which is now available. With GHC we would ensure that the infrastructure would persist, but there could still be maintenance issues if the GHC-API or the underlying representation of programs were to change.

The `syntax_tools` package for Erlang provides a very attractive solution for the tool writer because it gives an *abstract* interface to the underlying syntax trees which means that programs can be independent of the particular representation of the tree. It is also *extensible* in allowing additional attributes to be attached to AST nodes, and we use this to record detailed position and static semantic information.

An alternative would have been to build a framework for ourselves, as has been done by the RefactorErl group (Horváth *et al.*, 2008). This gives them complete control of the representation and processing of information, but comes with a maintenance cost: each time there is a change to the language then you need to upgrade the framework. It also has the disadvantage that some extensions to the language, such as the macro-processing system, either have to be reimplemented or worked around.

What should a tool support? Our tools aim to automate the application of a set of basic refactorings, and feedback and data gathered from usage suggests that this is a sensible approach. Of the refactorings supported, renaming and function extraction have proved to be the most commonly used ones.[9]

On the other hand, we have had requests from users to have help in directing refactoring effort, and this led to the developments reported in Sections 10 and 11. These give the user information – such as the presence of a clone and how it can be refactored away – which could be found by inspection but which would simply not be done that way because of the degree of details involved. Nevertheless, the underlying support for the transformations

[9] We are reminded of the 1980s debate about 'RISC versus CISC' for microprocessor instruction sets: we definitely take the 'RISC' approach here.

is in the basic refactorings, and we see this as confirmation of the appropriateness of our approach.

Past and present obstacles. Obstacles in the past have included over-complicated or non-standard installation mechanisms and dependencies on other systems, such as cygwin on Windows platforms. Wrangler is now available with a Windows installer on Windows and makefiles on Unix systems. HaRe is available from the Hackage database.

The fact that we have not provided support for particular editors, such as XEmacs and Vim, has been seen as an obstacle to adoption. We have had to take pragmatic decisions about the cost/benefit of supporting particular editors, and Wrangler is now supported in XEmacs but not in Vim.

For a Haskell refactoring tool to be adopted widely, it must support GHC Haskell: whether or not this is desirable, it is plainly the case. At the time that HaRe was first designed using the internals of GHC was not seen as a sensible route to take, but clearly now this makes the best sense. We have looked at 'porting' HaRe to GHC (and the GHC-API), but the degree to which HaRe depends on the details of the Programatica front end makes this a delicate and difficult task. Instead, it makes more sense to reimplement HaRe in GHC, using the experience gained in building it and Wrangler.

Principles versus engineering. Our initial aim in investigating refactoring for functional languages was to understand more clearly what refactoring means in this context. However, we have found that much of our work has not focussed on principles but rather on the effective *engineering* of these systems to embody the aspirations of Section 4.3. Particularly time-consuming have been *integration* with Eclipse, building scalable and incremental clone detection infrastructure and dealing with layout and formatting in the tools.

Refactoring and testing. We have found particularly fruitful links between refactoring and testing in our work with Wrangler. Test code is particularly prone to contain code clones, and we have had success in applying Wrangler's clone detection and removal in this area (Li *et al.*, 2009). One of the key problems in using QuickCheck-style tools is to find properties of systems. Our work builds on clone detection to identify properties from existing test suites (Li *et al.*, 2011). Finally, we see much scope for refactoring tests within test frameworks like EUnit (Carlsson & Rémond, 2006).

Trust and verification. For a program transformation tool to be used in practice, it must be trustworthy, and not to break existing code. The state of the art in the OO world (Fowler, 1999) is to expect the user to verify each instance of a refactoring by regression testing code before and after the refactoring is applied as well as to test refactoring tools as outlined in Section 9.4. Is it possible to do better for functional programming languages?

We can answer this in a number of ways. Firstly, if we write our tools in a functional language, then we can make the general claim that because functional languages are particularly well suited to manipulating language data structures and constructs, then our refactoring programs are more amenable to informal verification by inspection. (This kind of evidence might well be used in support of compilers written in functional languages, too.)

Secondly, we could argue that formal verification of refactorings for functional languages is itself possible. We have indeed shown how to formalise refactorings in Li & Thompson (2005) and investigated the foundations of formal verification of refactorings (Sultana & Thompson, 2008). In the latter case we modelled a renaming refactoring in a lambda calculus with name capture, and showed that under the appropriate preconditions the binding structure was not changed (and so, in particular, no capture took place). While this and other works, e.g. Garrido & Meseguer (2006), Carvalho Júnior *et al.* (2007), Ubayashi *et al.* (2008) and Yin *et al.* (2009), show that verification is possible in principle, practical verification of refactoring engines for Haskell or Erlang remains some way off, and is predicated on the mechanisation of the static and dynamic semantics as well as the module structure and type systems of the languages in question.[10]

A generic general-purpose refactoring tool? We have the experience of building general-purpose refactoring tools for two quite different functional languages, and so a natural question to ask is whether these two tools could be replaced by a single generic, general-purpose tool which works with a range of languages. Our answer is 'yes and no': 'yes' to the idea that generic tools such as StrategoXT (Bravenboer *et al.*, 2008) can be used very effectively to implement particular transformations for particular languages (as was done, for example, with the Y2K problem some years ago), but 'no' to the idea that general-purpose tools like Wrangler and HaRe could be subsumed as part of a single, generic and general-purpose system. Why do we take this negative position?

For a tool to be acceptable to users of a set of programming languages, the tool needs to cover all aspects of each of the languages, including their syntax, static semantics, type system, module structure, pre-processor, layout and comment conventions. For this to be effective, we need to be able to build a generic front-end for a compiler: not just a parser generator, but a fully featured representation of static semantics and types. This in turn will require internal representations that are flexible and powerful enough to represent concepts from different languages. To give a couple of illustrations from our experience, this will include modelling the multiple binding occurrences of Erlang variables, Haskell's extensions in GHC, including type-level programming, as well as the syntactic peculiarities of the two languages.

Despite the promise of attribute grammar and other formalisms, there has been no progress in building a single compiler for all programming languages (for the reasons above and more), and we anticipate equally large obstacles to it happening for refactoring engines. Moreover, in building tools for Haskell and Erlang we have been able to reuse substantial parts of the language infrastructure written in the languages themselves; this would not be possible in the generic case where all would have to be built from scratch in the 'brave new world' of the generic representation.

15 Conclusions

We have presented an overview of refactoring functional programs, and shown how different the process – and its implementation – can be for two representative functional

[10] See Alglave *et al*. (2011) for another perspective on this from the C verification community.

languages, Haskell and Erlang, and seen that because of the differences of the languages the tools are substantially different.

A number of refactorings that we have implemented – particularly the *structural* ones – are similar to OO refactorings; other OO refactorings which move methods and attributes around the inheritance hierarchy do not have direct equivalents in the functional paradigm. Finally, some refactorings of concurrent systems are supported more easily in a functional context.

Our work with Wrangler has underlined that it is important not only to implement the basic refactorings but also to provide *decision support tools*, such as clone detection and module analysis, that can guide the application of the tool. This was particularly evident in case studies with working software engineers (Li *et al.*, 2009).

Wrangler's API and DSL give users the ability to write refactorings themselves without having to become familiar with the intricacies of Wrangler's implementation. We see this as the way of the future, particularly when users begin to use online repositories to share their refactorings with each other.

We are continuing to work on refactoring for functional languages, and would identify the following challenges:

- Build a tool for refactoring GHC Haskell programs based on the GHC-API and other aspects of the GHC toolchain.
- Identify and implement a set of high-level *data refactorings*. A challenging problem here would be transform a list-processing program into a 'Hughes list' version, in which lists are represented by functions, and where concatenations are more efficiently implemented. What makes this question a challenge is that it is not clear what the scope of the transformation should be: are all lists in the system to be thus transformed, say?
- Identify and implement further decision support tools for Wrangler and HaRe users.

Acknowledgments

Thanks to Claus Reinke who worked with us at the start of this project, Chris Brown whose thesis and more recent work have extended the refactorings, Nik Sultana who looked at the foundations of verification for functional refactorings, György Orosz who built and maintained the Eclipse embedding of Wrangler and other students from ELTE who contributed through their Master's degree projects. We also want to thank the contributors to the Wrangler project at https://github.com/RefactoringTools/wrangler.

The development of HaRe and Wrangler was supported by the UK Engineering and Physical Sciences Research Council (grants GR/R75052/01, EP/C524969/1) and Wrangler was further supported by the European Commission Framework 7 programme (ProTest, grant no. 215868; RELEASE, 287510), and we are very grateful to them for their support, and to our project partners for their collaboration. Finally, we would like to thank the anonymous reviewers of this paper for their detailed and incisive comments.

References

Alglave, J., Donaldson, A. F., Kroening, D. & Tautschnig, M. (2011) Making software verification tools really work. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis (ATVA'11)*. Berlin, Germany: Springer-Verlag.

Armstrong, J. (2007) *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf.

Baker, B. S. (1992) A program for identifying duplicated code. *Comput. Sci. Stat.* **24**, 49–57.

Baker, B. S. (1995) On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, L. Wills, P. Newcomb & E. Chikofsky (eds), Washington, DC, USA: IEEE Computer Society Press, pp. 86–95.

Bravenboer, M., Kalleberg, K. T., Vermaas, R. & Visser, E. (2008) Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.* **72**, 52–70.

Brown, C. (2008) *Tool Support for Refactoring Haskell Programs*. PhD thesis, School of Computing, University of Kent, Kent, UK.

Brown, C., Loidl, H.-Wo. & Hammond, K. (2011) Refactoring parallel Haskell programs. *Symposium on Trends in Functional Programming (TFP'11)*, Madrid, Spain, May 2011, LNCS 7193.

Brown, C. & Thompson, S. (2010) Clone detection and elimination for Haskell. In *Partial Evaluation and Program Manipulation (PEPM'10),* Gallagher, J. & Voigtlander, J. (eds), New York, NY: ACM Press, pp. 111–120.

Bulychev, P. & Minea, M. (2008) Duplicate code detection using anti-unification. In *Spring Young Researchers Colloquium on Software Engineering (SYRCoSE)*, Saint-Petersburg, Russia, May 29–30, pp. 51–54.

Burstall, R. & Darlington, J. (1975) Some transformations for developing recursive programs. In *Proceedings of the International Conference on Reliable Software*. New York, NY: ACM, pp. 465–472.

Carlsson, R. & Rémond, M. (2006) EUnit: A lightweight unit testing framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (ERLANG '06)*, Portland, Oregon, USA.

Carvalho Júnior, A., Silva, L. & Cornélio, M. (2007) Using CafeOBJ to mechanise refactoring proofs and application. *Electron. Notes Theor. Comput. Sci.* **184** (July), 39–61.

Chassell, R. J. (2004) *An Introduction to Programming in Emacs Lisp*. Boston, MA: Free Software Foundation.

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, Montréal, Canada. New York, NY: ACM, pp. 268–279.

Daniel, B., Dig, D., Garcia, K. & Marinov, D. (2007) Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*, Dubrovnik, Croatia. New York, NY: ACM, pp. 185–194.

Darlington, J. (1982) Program transformation. In *Functional Programming and Its Applications*, Darlington, J., Henderson, P. & Turner, D. A. (eds), Cambridge, UK: Cambridge University Press.

Drienyovszky, D., Horpacsi, D. & Thompson, S. (2010) QuickChecking refactoring tools. In *Proceedings of the 2010 ACM SIGPLAN Erlang Workshop (Erlang'10)*, Baltimore, Maryland, USA. Fritchie, S. L. & Sagonas, K. (eds). New York, NY: ACM SIGPLAN, pp. 75–80.

Erlang E-learning. (2011) *Online e-Learning System.* Available at: *https://elearning.erlang-solutions.com.* Accessed 6 September 2013.

ErlIDE. (n.d.) The Erlang IDE. Available at: *http://erlide.sourceforge.net/.* Accessed 6 September 2013.

Evans, W., Fraser, C. & Ma, F. (2008) Clone detection via structural abstraction. In *The 14th Working Conference on Reverse Engineering*, IEEE Computer Society Press, Washington, DC, USA. Vancouver, BC, Canada, pp. 150–159.

Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Boston, MA: Addison-Wesley.

Gallardo, D. (n.d.) *Refactoring for Everyone: How and Why to use Eclipse's Automated Refactoring Features*. Available at: http://www.ibm.com/developerworks/library/os-ecref/. Accessed 6 September 2013.

Garrido, A. & Meseguer, J. (2006) Formal specification and verification of Java refactorings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06)*, Philadelphia, PA, USA. Washington, DC: IEEE Computer Society.

Ge, X., DuBose, Q. L. & Murphy-Hill, E. (2012) Reconciling manual and automatic refactoring. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012*), Zurich, Switzerland. New York, NY: IEEE Press, pp. 211–221.

GHC API. (n.d.) *GHC Commentary*. Available at: http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/API. Accessed 6 September 2013.

Göde, N. & Koschke, R. (2009) Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR'09)*, Kaiserslautern, Germany.

Griswold, W. G. (1991) *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

Griswold, W. G. & Notkin, D. (1993) Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.* **2**, 228–269.

Hackage. (2010) *HackageDB*. Available at: http://hackage.haskell.org/. Accessed 6 September 2013.

Hallgren, T. (2003) Haskell tools from the programatica project (Demo Abstract). *In ACM Sigplan Haskell Workshop*, Uppsala, Sweden. New York, NY: ACM Press.

HaRe. (n.d.) *The HaRe system*. Available at: http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html. Accessed 6 September 2013.

Haskell Platform. (2010) *The Haskell Platform*. Available at: http://hackage.haskell.org/platform/. Accessed 6 September 2013.

Hills, M., Klint, P. & Vinju, J. J. (2012) Scripting a refactoring with Rascal and Eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools (WRT '12)*, Rapperswil, Switzerland. New York, NY: ACM, pp. 40–49.

Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A. N., Nagy, T., Tóth, M. & Király, R. (2008) Building a refactoring tool for Erlang. In *Workshop on Advanced Software Development Tools and Techniques (WASDETT 2008)*, Paphos, Cyprus.

Hughes, J. & Peyton Jones, S. (eds). (1999) *Report on the Programming Language Haskell 98*. Available at: `http://www.haskell.org/onlinereport/`. Accessed 6 September 2013.

Jiang, L., Misherghi, G., Su, Z. & Glondu, S. (2007) DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, Minneapolis, MN. Washington, DC: IEEE Computer Society, pp. 96–105.

Katić, M. & Fertalj, K. (2009) Towards an appropriate software refactoring tool support. In *Proceedings of the 9th WSEAS International Conference on Applied Computer Science* (ACS'09), Genova, Italy. Stevens Print, WI: World Scientific and Engineering Academy and Society (WSEAS), pp. 140–145.

Kitlei, R., Bozó, I., Kozsik, T., Tejfel, M. & Tóth, M. (2010) Analysis of Preprocessor Constructs in Erlang. In *Proceedings of the 9th ACM SIGPLAN Erlang Workshop*, Baltimore, Maryland, USA, pp. 45–55.

Kitlei, R., Lövei, L., Tóth, M., Horváth, Z., Kozsik, T., Király, R., Bozó, I., Hoch, C. & Horpácsi, D. (2008 November) Automated syntax manipulation in RefactorErl. In *14th International Erlang/OTP User Conference*, Stockholm, Sweden.

Komondoor, R. & Horwitz, S. (2001) Tool Demonstration: Finding Duplicated Code Using Program Dependences, *Lecture Notes in Computer Science*, 2028. New York, NY: LNCS.

Lahoda, J., Bečička, J. & Ruijs, R. B. (2012) Custom declarative refactoring in NetBeans: Tool demonstration. In *Proceedings of the Fifth Workshop on Refactoring Tools* (WRT '12), Rapperswil, Switzerland. New York, NY: ACM, pp. 63–64.

Lämmel, R. (2000) Reuse by program transformation. In *Selected Papers from the 1st Scottish Functional Programming Workshop*, Functional Programming Trends series, vol. 1, Michaelson, G. & Trinder, P. (eds). Bristol, UK: Intellect, pp. 143–152.

Lämmel, R. & Visser, J. (2003) A Strafunski Application Letter, In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages. PADL'03*, Springer Verlag, Berlin and New York.

Leitão, A. P. T. de M. C. (2002) A formal pattern language for refactoring of Lisp programs. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR '02)*, Budapest, Hungary. Washington, DC: IEEE Computer Society.

Li, H. (2006) *Refactoring Haskell programs*. PhD thesis, School of Computing, University of Kent, Canterbury, Kent, UK.

Li, H., Lindberg, A., Schumacher, A. & Thompson, S. (2009) *Improving Your Test Code with Wrangler*. Tech. Rept. 4-09, School of Computing, University of Kent, Canterbury, Kent, UK.

Li, Z., Lu, S. & Myagmar, S. (2006) CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.* **32**(3), 176–192.

Li, H. & Thompson, S. (2005) Formalisation of Haskell refactorings. In *Trends in Functional Programming*, M. van Eekelen & K. Hammond (eds). Tallinn, Estonia. Bristol, UK: Intellect, pp. 95–110.

Li, H. & Thompson, S. (2006) A comparative study of refactoring Haskell and Erlang programs. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, September 27–29, Sheraton Society Hill, Philadelphia, PA, Di Penta, M. & Moonen, L. (eds). New York, NY: IEEE, pp. 197–206.

Li, H. & Thompson, S. (2007) Testing Erlang refactorings with quickcheck. In *19th International Symposium on Implementation and Application of Functional Languages (IFL 2007)*, Freiburg, Germany. New York, NY: LNCS.

Li, H. & Thompson, S. (2009a). Clone detection and removal for Erlang/OTP within a refactoring environment. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, Savannah, GA, USA.

Li, H. & Thompson, S. (2009b) Testing-framework-aware refactoring. In *3rd ACM Workshop on Refactoring Tools*, Orlando, FLA, USA, pp. 1–4.

Li, H. & Thompson, S. (2010a) Refactoring support for modularity maintenance in Erlang. In *10th IEEE International Working Conference on Source Code Analysis and Manipulation*, J. Vunju & C. Marinescu (eds), Timisoara, Romania. New York, NY: IEEE Computer Society, pp. 157–166.

Li, H. & Thompson, S. (2010b). Similar code detection and elimination for Erlang programs. In *Twelfth International Symposium on Practical Aspects of Declarative Languages (PADL10)*, Madrid, Spain.

Li, H. & Thompson, S. (2011a). *A User-Extensible Refactoring Tool for Erlang Programs*. Tech. Rept. 2011-04, School of Computing, University of Kent, Canterbury, Kent, UK.

Li, H. & Thompson, S. (2011b). Incremental code clone detection and elimination for Erlang programs. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*

(FASE'11), D. Giannakopoulou & F. Orejas (eds), Saarbrücken, Germany. New York, NY: Springer.

Li, H. & Thompson, S. (2012a). A domain-specific language for scripting refactoring in Erlang. In *Fundamental Approaches to Software Engineering (FASE 2012)*, de Lara, J. & Zisman, A. (eds). Lecture Notes in Computer Science, Tallinn, Estonia. New York, NY: Springer, vol. 7212, pp. 294–297.

Li, H. & Thompson, S. (2012b). Automated API migration in a user-extensible refactoring tool for Erlang programs. In *Automated Software Engineering (ASE'12*), Menzies, T. & Saeki, M. (eds), Essen, Germany. New York, NY: IEEE Computer Society.

Li, H. & Thompson, S. (2012c). Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools (WRT '12)*, Rapperswil, Switzerland. New York, NY: ACM, pp. 32–39.

Li, H., Thompson, S. & Arts, T. (2011) Extracting properties from test cases by refactoring. In *Refactoring and Testing Workshop (RefTest 2011)*, S. Counsell (ed). New York, NY: IEEE Digital Library.

Li, H., Thompson, S., Orosz, G. & Töth, M. (2008) Refactoring with wrangler, updated. In *ACM SIGPLAN Erlang Workshop 2008*, Victoria, British Columbia, Canada.

Lövei, L. (2009) Automated module interface upgrade. In *Proceedings of the 2009 ACM SIGPLAN Erlang Workshop*, Edinburgh, Scotland. New York, NY: ACM, pp. 11–21.

Lövei, L., Hoch, C., Köllő, H., Nagy, T., Nagyné-Víg, A., Horpácsi, D., Kitlei, R. & Király, R. (2008) Refactoring module structure. In *Proceedings of the 7th ACM SIGPLAN Erlang Workshop*, Victoria, British Columbia, Canada. New York, NY: ACM, pp. 83–89.

Marlow, S. (ed). (2010) *Haskell 2010 language report*. Available at: http://www.haskell.org/onlinereport/haskell2010/. Accessed 6 September 2013.

Mens, T. & Tourwé, T. (2004) A survey of software refactoring. *IEEE Trans. Softw. Eng.* **30**, 126–139.

Meszaros, G. (2007) *xUnit Test Patterns: Refactoring Test Code*. Boston, MA: Addison-Wesley.

Mitchell, N. (2011) *HLint Manual*. Available at: http://community.haskell.org/~ndm/hlint/. Accessed 6 September 2013.

Murphy-Hill, E., Parnin, C. & Black, A. P. (2009) How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, Vancouver, British Columbia, Canada. Washington, DC: IEEE Computer Society, pp. 287–297.

NetBeans. (n.d.) *Wiki entry on refactoring*. Available at: http://wiki.netbeans.org/Refactoring. Accessed 6 September 2013.

Nguyen, T. T., Nguyen, H. A., Al-Kofahi, J. M., Pham, N. H. & Nguyen, T. N. (2009) Scalable and incremental clone detection for evolving software. In *International Conference on Software Maintenance (ICSM'09),* Edmonton, Canada.

Opdyke, W. F. (1992) *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL.

ReSharper. (n.d.) *Introductory web page.* Available at: http://www.jetbrains.com/resharper/. Accessed 6 September 2013.

Roberts, D., Brant, J. & Johnson, R. (1997) A refactoring tool for smalltalk. *Theory Pract. Object Syst.* **3**(4), 253–263.

Roy, C. K., *et al.* (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* **74**(7), 470–495.

Sagonas, K. (2007) Detecting defects in Erlang programs using static analysis. In Proceedings of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming *(PPDP '07)*, Wrocław, Poland. New York, NY: ACM, pp. 37–37.

Sagonas, K. & Avgerinos, T. (2009) Automatic refactoring of Erlang programs. In *Principles and Practice of Declarative Programming (PPDP'09)*, Coimbra, Portugal. New York, NY: ACM, pp. 13–24.

Schaefer, M. & de Moor, O. (2010) Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, Reno, Nevada, USA. New York, NY: ACM, pp. 286–301.

Soares, G. (2012) Automated behavioral testing of refactoring engines. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*, Tucson, Arizona, USA. New York, NY: ACM, pp. 49–52.

Sommerlad, P. (ed). (2012) *Proceedings of the Fifth Workshop on Refactoring Tools (WRT '12)*, Rapperswil, Switzerland. New York, NY: ACM.

Sultana, N. & Thompson, S. (2008) Mechanical verification of refactorings. In *Workshop on Partial Evaluation and Program Manipulation*, San Francisco, California, USA. New York, NY: ACM SIGPLAN.

Thompson, S. & Reinke, C. (2003) A case study in refactoring functional programs. *Brazilian Symposium on Programming Languages*, Ouro Preto, MG, Brazil, May 28–30.

Tóth, M. & Bozó, I. (2012) Static analysis of complex software systems implemented in Erlang. In *Central European Functional Programming Summer School (CEFP 2011), Revisited Selected Lectures*, Lecture Notes in Computer Science, vol. 7241. New York, NY: Springer-Verlag, pp. 451–514.

Tullsen, M. (2002) *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, Yale, CT.

Ubayashi, N., Piao, J., Shinotsuka, S. & University, T. (2008) Contract-based verification for aspect-oriented refactoring. In *1st International Conference on Software Testing, Verification, and Validation, 2008*, Lillehammer, Norway. New York, NY: IEEE Press.

Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P. & Johnson, R. E. (2012) Use, disuse, and misuse of automated refactorings. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland. New York, NY: IEEE Press, pp. 233–243.

Yin, X., Knight, J. & Weimer, W. (2009) Exploiting refactoring in formal verification. In *IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)*, Estoril, Lisbon, Portugal. New York, NY: IEEE Press.