A Comparative Study of Refactoring Haskell and Erlang Programs

Huiqing Li
Computing Laboratory
University of Kent
H.Li@kent.ac.uk

Simon Thompson
Computing Laboratory
University of Kent
S.J.Thompson@kent.ac.uk

Abstract

Refactoring is about changing the design of existing code without changing its behaviour, but with the aim of making code easier to understand, modify, or reuse. Taking Haskell and Erlang as examples, we investigate the application of refactoring techniques to functional programs, and building tools for supporting interactive refactoring.

Although both Haskell and Erlang are general-purpose functional programming languages, they have many differences in their language design and programming idioms. As a result, program refactoring in the two languages has much in common, but also considerable differences. This paper makes this comparison, and in particular looks in more detail at the refactorings applicable in each language, the program analysis required by typical refactorings, and at tool support for refactoring Haskell and Erlang programs.

1. Introduction

Refactoring [4] is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (nor remove) any bugs. Separating general software updates into functionality changes and refactorings has well-known benefits. While it is possible to refactor a program by hand, tool support is invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Refactoring tools can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the refactoring transformation itself, making the process less painful and error-prone.

Our project 'Refactoring Functional Programs' [10], has developed the Haskell Refactorer, HaRe [7], providing support for refactoring Haskell programs. HaRe is a mature tool covering the full Haskell 98 standard, including "notoriously nasty" features such as monads, and is integrated with the two most popular development environments for Haskell programs: Vim and (X)Emacs. HaRe refactorings apply equally well to single- and multiple-module projects. HaRe is itself implemented in Haskell.

Haskell layout style tends to be idiomatic and personal, especially when a standard layout is not enforced by the program editor, and so needs to be preserved as much as possible by refactorings. HaRe does this, and also retains comments, so that users can recognise their source code after a refactoring. The current release of HaRe supports 24 refactorings, and also exposes an API [5] for defining Haskell refactorings and program transformations.

The refactorings supported by HaRe fall into three categories: *structural* refactorings affecting the names, scopes and structure of the entities defined in a program; *module* refactorings affecting the imports and exports of modules and the definitions contained in them; and *data-oriented* refactorings of data types. All these refactorings have been successfully applied to multiple module systems containing tens of thousands of lines of code.

'Formally-Based Tool Support for Erlang Development' [3] is a joint research project between Universities of Kent and Sheffield, to build a variety tools to support working Erlang programmers. The aspect relevant here is the construction of a refactoring tool which can be used by Erlang programmers in practice. This work allows us to continue our investigation of the application of refactoring techniques to the functional programming paradigm.

Both Haskell and Erlang are general-purpose functional programming languages, but they also have many differences. Haskell [12] is a lazy, statically typed, purely functional language featuring higher-order functions, polymorphism, type classes, and monadic effects. Erlang [1] is a strict, dynamically typed functional programming language with built-in support for concurrency, communication, distribution, and fault-tolerance. In contrast to Haskell, which arose from an academic initiative, Erlang was developed in the Ericsson Computer Science Laboratory, and has been actively used in industry both within Ericsson and beyond.

We have established the architecture of the Erlang refactorer, the *Wrangler*, and two refactorings, *rename an identifier* and *generalise a function definition*, have been imple-

```
module Fact(fac) where
fac :: Int -> Int
fac 0 = 1
fac n | n>0 = n * fac(n-1)
```

Figure 1. Factorial in Haskell

```
-module (fact).
-export ([fac/1]).
fac(0) -> 1;
fac(N) when N > 0 -> N * fac(N-1).
```

Figure 2. Factorial in Erlang

mented in the Wrangler. While we are still in the early stage of the project, it is already clear to us that implementing refactorings within the Wrangler is by no means a simple reimplementation of a collection of refactorings for Haskell.

In this paper, we compare the similarities and differences between refactoring Haskell and Erlang programs based on our experience so far. The comparison is carried our from three perspectives: the refactorings supported by each language; the program analysis necessitated by typical refactorings; and the implementation of HaRe and the Wrangler.

The rest of this paper is organized thus: in Section 2, we give a brief description of Haskell and Erlang; in Section 3, we introduce refactoring and illustrate how it can be part of the program development process. In Section 4, we discuss tool support for the refactoring process; in Section 5, we discuss and compare some typical refactorings for Haskell and Erlang; in Section 6, we compare the program analysis required by typical refactorings in the two languages. The implementations of HaRe and the Wrangler are discussed in Section 7; Section 8 completes the paper with a conclusion and some observations on future work.

2. Haskell and Erlang

Haskell [12] is a typical of many modern functional languages. It manifests features such as higher-order functions, lazy evaluation, equations, and pattern matching over algebraic data types, a type system with Hindley-Milner type inference and type classes, overloading, monadic programming, and a module system. Haskell has evolved continuously since its first publication. The current standard version is Haskell 98, and defined in *Haskell 98 Language and Libraries: the Revised Report* [12]. A Haskell program is a collection of modules. A module defines a collection of values, data types, type synonyms, classes, etc. A Haskell module may import definitions from other modules, and re-exports some of them and its own definitions, making them available to other modules. Figure 1 shows a Haskell module containing the definition of the factorial function.

Erlang is a strict, dynamically typed, functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance, and dynamic code reloading [1]. Erlang's elementary data types are atoms, numbers (integers and floats), process identifiers, references, binaries, and ports; compound data types are tuples and lists.

Erlang also comes with a module system. An Erlang program typically consists of a number of modules, each of which defines a collection of functions. Only functions exported explicitly may be called from other modules, and a module can only export functions which are defined in the module itself.

Erlang has built-in support for concurrency. Processes and communication between processes are fundamental concepts in Erlang. A process is a self-contained, separate unit of computation which executes concurrently with other processes in the system. The primitives **spawn**, "**!**" (send) and **receive** allow a process to create a new process and to communicate with other processes through asynchronous message passing.

The Erlang language itself is small, but it comes with libraries containing a large set of built-in functions. Erlang has also been extended by the Open Telecom Platform (OTP) middleware platform, which provides a number of ready-to-use components and design patterns, such as finite state machines, generic servers, etc, embodying a set of design principles for Erlang systems.

Figure 2 shows an Erlang module containing the definition of the factorial function. In this example, `fac/1` denotes the function `fac` with arity of 1. In Erlang, a function name can be defined with different arities, and the same function name with different arities represents entirely different functions computationally.

3. Functional Refactoring

3.1 An Example

Figures 3 - 7 illustrate how refactoring techniques can be used in the program development process. This example is written in Erlang, however the same ideas apply to Haskell programs. The example presented here is small-scale, but it is chosen to illustrate aspects of refactoring which can scale to larger programs and multi-module systems.

In Figure 3, the function `printList/1` has been defined to print all elements of a list to the standard output.

```
-module (sample).
-export([printList/1]).

printList([H|T]) ->
    io:format("~p\n", [H]),
    printList(T);
printList([]) -> true.
```

Figure 3. The initial program

```
-module (sample).
-export([printList/1, broadcast/1]).

printList([H|T]) ->
    io:format("~p\n", [H]),
    printList(T);
printList([]) -> true.

broadcast([H|T]) ->
    H ! "The message",
    broadcast(T);
broadcast([]) -> true.
```

Figure 4. Adding a new function naïvely

Next, suppose the user would like to add another function, `broadcast/1`, which broadcasts a message to a list of processes. `broadcast/1` has a very similar structure to `printList/1`, as they both iterate over a list doing something to each element in the list. Naïvely, the new function could be added by copy, paste, and modification as shown in Figure 4. However, a *refactor then modify* strategy, as shown in Figures 5 - 7, would make the resulting code easier to maintain and reuse.

Figure 5 shows the result of generalising the function `printList` on the sub-expression `io:format("~p/n", [H])`. The expression contains the variable `H`, which is only in scope within the body of `printList`. Instead of generalising over the expression itself, the transformation is achieved by first abstracting over the free variable `H`, and by making the generalised parameter a *function* `F`. In the body of `printList` the expression `io:format("~p/n", H])` has been replaced with `F` applied to the local variable `H`.

The arity of the `printList` has thus changed; in order to preserve the interface of the module, we create a new function, `printList/1`, as an application instance of `printList/2` with the first parameter supplied with the function expression:

```
-module (sample).
-export([printList/1]).

printList(L) ->
    printList(fun(H) ->
        io:format("~p\n", [H]) end, L).

printList(F,[H|T]) ->
    F(H),
    printList(F, T);
printList(F,[]) -> true.
```

Figure 5. The program after generalisation

```
-module (sample).
-export([printList/1]).

printList(L) ->
    forEach(fun(H) ->
        io:format("~p\n", [H]) end, L).

forEach(F,[H|T]) ->
    F(H),
    forEach(F, T);
forEach(F,[]) -> true.
```

Figure 6. The program after renaming

`fun(H) -> io:format("~p/n", [H]) end`. Note that this transformation gives `printList` a functional argument, thus making it a characteristically 'functional' refactoring.

Figure 6 shows the result of renaming `printList/2` to `forEach/2`. The new function name reflects the functionality of the function more precisely. In Figure 7, function `braodcast/1` is added as another application instance of `forEach/2`.

Generalise a function definition and *rename an identifier* are typical structural refactorings, supported by both HaRe and the Wrangler.

3.2 Behaviour Preservation

The essential criterion for behaviour preservation is that given the same input value(s), the program should produce the same output value(s) before and after the refactoring.

The value of a Haskell program is the value of the `main` function defined in the `Main` module. Therefore, for behaviour preservation, we require that given the same input value(s), the identifier `main` defined in the `Main` module of

```
-module (sample).
-export([printList/1, broadcast/1]).

printList(L) ->
    forEach(fun(H) ->
        io:format("~p\n", [H]) end, L).

broadcast(Pids)->
    forEach(fun(H) ->
        H ! "The message" end, Pids).

forEach(F,[H|T]) ->
    F(H),
    forEach(F, T);
forEach(F,[]) -> true.
```

Figure 7. The program after adding a function

the program under refactoring should produce the same output value(s) before and after the refactoring. The semantics of other functions defined in the program could be changed after a refactoring, as long as the change does not affect the value of `main`.

No such `Main` module is mandatory for an Erlang program. An Erlang program is more like a library with each module exporting a number of functions. In this case, it is reasonable to require refactorings to preserve the functionalities of those functions exported by each module. This is also reflected by the example shown in Figure 5 where an extra function has been added but the interface of the module is not changed. However, if the user feels that this is too restrictive, the refactorer should allow the module interface to be changed, and automatically compensate the change throughout the system in question.

3.3 Semantics and Transformation

Refactorings are not simply syntactic. In order to preserve the functionality of a program, refactorings require awareness of various aspects of the semantics of the program including types and module structure and most importantly the *static semantics* of the program: that is the scope of definitions, the binding structure of the program (the association between the use of an identifier and its definition), the uniqueness of definitions and so forth.

Each refactoring comes with a set of side conditions, which embody when a refactoring can be applied to a program without changing its meaning. Our experience of building refactoring tools so far shows that for most refactorings, the side-condition analysis is more complex than the program transformation part. Taking a concrete example, among the side conditions for renaming an identifier could be:

The existing binding structure should not be affected. No binding for the new name may intervene between the binding of the old name and any of its uses, since the renamed identifier would be captured *by the renaming. Conversely, the binding to be renamed must not intervene between bindings and uses of the new name.*

The above side-conditions apply to both Haskell and Erlang programs, and indeed to most other programming languages. However, each programming language may also impose its own particular constraints on this refactoring. For example, in a Haskell program, the `main` function defined in the `Main` module should not be renamed. In an Erlang program using the OTP library, a user should not rename certain functions exported by a call-back module, for instance.

One difference between Haskell and Erlang emerges at this point. Conditions on Haskell refactorings can generally be checked at compile time; the more dynamic nature of Erlang means that some necessary conditions can only decided at run-time; we return to this point below.

4. Tool Support for Refactorings

Although it is possible to refactor a program manually, it would be very tedious and error-prone to refactor large programs this way. In this case, interactive tool support for refactoring is invaluable, as it is more reliable and allows refactorings to be done and undone very easily.

A refactoring tool needs to get access to both the syntactic and static semantic information of the program under refactoring. While detailed implementation techniques might be different, most refactoring tools go through the following process: first transform the program source to some internal representation, such as an abstract syntax tree (AST); then analyse the program to extract the necessary static semantic information, such as the binding structure of the program, type information and so forth. After that, program analysis is carried out based on the internal representation of the program and the static semantics information to validate the side-conditions of the refactoring. If the side-conditions are not satisfied, the refactoring process stops and the original program is unchanged, otherwise the internal representation of the program is transformed according to the refactoring. Finally, the transformed representation of the program need to be presented to the programmer in program source form, with comments and the original program appearance preserved as much as possible.

5 Haskell vs. Erlang Refactoring

A growing catalogue of Haskell refactorings is available from our project webpage [10], and we are still in the process of collecting practically useful Erlang refactorings. In this paper, we select a small number of representative refactorings to illustrate how different language features and programming idioms can exhibit different refactoring opportunities and challenges.

5.1 Common Refactorings

These refactorings mainly concern the name and scope of the entities defined in a program, the structure of definitions and the module system. Apart from the previously mentioned *renaming an identifier* and *generalising a definition*, some other refactorings are:

- *Removing unused function definitions or parameters*.
- *Introducing a new definition* to represent an identified expression.
- *Swapping arguments* of a function definition.
- *Lifting a locally defined function* to top level of the module. In Erlang, a function name needs to be supplied for the lifted function.
- *Unfolding a definition* by replacing an identified occurrence of the left-hand side of a definition with the instantiated right-hand side.
- *Moving a definition from one module to another*.
- *Splitting a large module into two*.

5.2 Haskell-specific Refactorings

In Haskell, *type classes* provide a structured way to control overloading. They allow the declaration of types as instances of the class by providing definitions of the overloaded *operations* associated with a class [12]. One of the type class related refactorings is:

- *Introduce overloading* by identifying a type, a class definition and a collection of functions over that type which are to form the body of the instance declaration.

Haskell is a statically typed language and allows the declaration of algebraic data types. There is a collection of datatype related refactorings which are also Haskell specific. A couple of examples are:

- *Naming a type using* `type` by identifying uses of a type in a particular way and making them instances of a type synonym.
- *From concrete to abstract data type*. This refactoring turns an identified concrete data type into an abstract data type. A concrete data type exposes the representation of the data type, so that the users can get access to the data constructors defined in the data type and write pattern matching definitions, whereas an abstract data type hides the data representation from the users. Making a data type abstract allows changing the representation of the data type without affecting the client functions that use this data type. This refactoring is described in more detail in [14]. This refactoring is less-well suited for Erlang programs because Erlang does not allow the use of user-defined functions in guards.

5.3 Erlang-specific Refactorings

Built-in support for concurrency is one of the main features of Erlang. In an well-designed Erlang program, there should be a one-to-one mapping between the number of parallel processes and the number of truly parallel activities in the real world. The following refactoring allows to adjust the process structure in a program.

- *Introduce/remove concurrency* by introducing or removing concurrent processes so as to achieve a better mapping between the parallel processes and the truly parallel activities of the problem being solved. For example, using processes and message passing when a function call can be used instead is a bad programming practice, and this refactoring should help to eliminate the un-desired process and message passing with a function call.

While defensive-style programming is a good programming practice when a sequential programming language is used, non-defensive style programming is the right thing to do when programming with Erlang. Erlang's *worker/supervisor* error handling mechanism allows a clear separation of error recovery code and normal case code. In this mechanism, both *workers* and *supervisors* are processes, where *workers* do the job, and *supervisors* observe the *workers*. If a *worker* crashes, it sends an error signal to its *supervisor*.

- *From defensive-style programming to non-defensive style*. This refactoring helps to transform defensive-style sequential error-handling code written in Erlang into concurrent error handling, typically using supervisor trees.

Erlang programming idioms also expose various refactoring opportunities. Some examples are:

- *Transform a non-tail-recursive function to a tail-recursive function*. In Erlang, all servers must be tail-recursive, otherwise the server will consume memory until the system runs of it.

- *Remove import attributes*. Using import attributes makes it harder to directly see in what module a function is defined. Import attributes can be removed by using remote function call when a call of function defined in another module is needed.

- *From meta to normal function application* by replacing `apply(Module, Fun, Args)` with `Module:Fun(Arg1, Arg2,..., ArgN)` when the number of elements in the arguments, `Args`, is known at compile-time.

- *Refactoring non-OTP code towards an OTP pattern*. Doing this from pure Erlang code is going to be very challenging, but the whole transformation can be decomposed into a number of elementary refactorings, and each elementary refactoring brings the code a bit closer to the desired OTP pattern.

6 Analysis of Haskell and Erlang Programs

This section discusses and compares some typical static semantic analysis involved when refactoring Haskell and Erlang programs, and illustrates the fact that analysing Erlang programs is in general more complex than analysing Haskell programs in several respects.

6.1 The Binding Structure of Variables

Binding structure refers to the association of uses of identifiers with their definitions in a program. In general, a refactoring should not disrupt the existing binding structure. However, in a programming language which allows declarations of the same name across nested scopes, the binding structure can be very easily disrupted without leading to an incorrect program (which fails to compile). For instance, in the following Haskell example:

```
g y = f y + 17 where h z = z + 34
```

the free variable `f` refers to some function defined outside the definition of `g`. Renaming function `h` to `f` would bind the free variable `f` to the renamed definition. This kind of change of binding structure would not always be detected by the compiler, and can only be avoided by proper side-condition checking and transformation rules.

Both Haskell and Erlang allow static scoping of variables, in other words, matching a variable to its binding only requires analysis of the program text. However, Erlang still differs from Haskell in three different ways.

First, in Haskell, every occurrence of a variable in a pattern is a binding occurrence, whereas in Erlang the binding occurrence of a variable always occurs in a pattern, but a pattern may also contain applied (i.e. non-binding) occurrences of variables.

Secondly, in Haskell, all patterns must be linear, i.e. no variable may appear more than once in the same pattern, whereas in Erlang, non-linear patterns with multiple occurrences of the same variable are allowed.

Finally, in Haskell, a variable is only associated with one binding occurrence, whereas in Erlang, a variable may have more than one binding occurrence, due to that `case/receive` expressions in Erlang can export variables. For instance, in the following Erlang code:

```
bar(S) -> case S of  1 -> S1 = 3, S1;
                     _ -> S1 = 4, S1
          end,
          S1 *S1.
```

both branches of the case expression export the variable `S1`. The applied occurrences of `S1` after the case expression are bound to both pattern bindings in the case expression.

Allowing a variable to have more than one occurrence has impacts on a number of refactorings. The most obvious case is *renaming a variable*, in which case renaming one binding occurrence of a variable might also need another binding occurrence of the same variable name to be renamed.

6.2 Finding the Call-sites of a Function

The function call graph is essential for refactorings that change the arity of type of a function. These refactorings, in general, modify not only the function definition itself, but also all the call-sites in order to compensate for the changes to the function definition itself.

Both Haskell and Erlang allow higher-order function calls. A function call in Haskell has the form: $e_1 e_2$; and a function call in Erlang has the form: $e_0(e_1, e_2, ...e_n)$, or $e_m : e_0(e_1, e_2, ..., e_n)$, where e_m should evaluate to a module name. Unlike Haskell, Erlang syntactically does not allow partial application of functions.

Apart from these forms of function application, Erlang also allows meta-applications using the built-in function `apply/3`. `apply/3` takes the following form: `apply(Module, Function, Args)`, where `Module` represents a module name, `Function` represents a function name, and `Args` is a list of terms. `apply/3` applies `Function` in `Module` to `Args`, and the arity of the applied function is the length of `Args`.

`apply/3` is useful when the number of arguments is *not* known at compile-time. However, as Erlang allows defining functions with the same name but different arities, it also imposes some difficulties with refactorings. For example, in the following trivial example:

```
f (X) -> X.
f (X, Y) -> X + Y + 1.
foo(Args)-> erlang:apply(math, f, Args).
```

for a refactoring tool that does only static semantic analysis, it is not possible to decide whether `f/1` or `f/2` will be called by the function `foo`. This causes problems for some refactorings. Most obviously, renaming of `f/1` might need `f/2` to be renamed as well.

Another difference between Haskell and Erlang is that unlike in Haskell, where a function name is a variable, a function name in Erlang is an atom literal. This causes two problems with refactoring Erlang programs:

First, as atoms can be used as function names, module names, process names, or just normal literals, it may not be straightforward to see whether an atom refers to a function name of not. For instance, in the following function call:

```
start() -> spawn(ch1, ch1, [])
```

The first `ch1` refers to a module name, and the second `ch1` refers to a function name.

Secondly, since atoms can be created from lists using the `list_to_atom/1` function, and a list can be computed from other lists, this provides the possibility of composing function names at run time, which again makes finding the call-sites of a function difficult.

The refactorer should give a warning message when it is not possible to tell whether a function is called at a specific site of the program.

6.3 Strict / Non-strict; Pure/ Impure

Haskell is a non-strict programming language, which means that an expression is only evaluated when its value is needed. On the contrary, Erlang is a strict language, so that arguments are evaluated before function application.

For a pure functional programming language, like Haskell, changes of the evaluation order of expressions should not change the functionality of the program. However for a non-pure language such as Erlang, in which mutable stuff (message sends/receives and state-dependent responses) plays an important part in most large programs, changes of the evaluation order could easily change the functionality of the program. For instance, in the following example:

```
repeat(0) -> ok;
repeat(N) -> io:format ("Hello"),
             repeat(N-1).
f() -> repeat(5).
```

The function `repeat/1` echoes the string `"Hello"` a specified number of times, and function `f/1` echoes the string 5 times by calling `repeat/1` with the argument `5`.

Naïvely generalising function `repeat/1` on the sub-expression `io:format ("Hello")` could produce:

```
repeat(A, 0) -> ok;
repeat(A, N) -> A,
                repeat(A,N-1).
f() -> repeat(io:format("hello"), 5).
```

In the above produced code, expression `io:format ("Hello")` will be evaluated with a returned value `ok`, before the body of the function `repeat/2` is executed. As a result, `f/0` only echoes the string `"Hello"` once instead of 5 times.

The above transformation changes the functionality of `f/1`. In order to preserve the behaviour of the function `f/1`, when generalising a function on a sub-expression with side-effect, we have to wrap this sub-expression in a function expression, and hence to delay the evaluation of the actual argument, as shown below.

```
repeat(A, 0) -> ok;
repeat(A, N) -> A(),
                repeat(A,N-1).
f() ->
  repeat(fun()->io:format("Hello") end,5).
```

It is the refactorer's responsibility to decide whether an expression has side-effects or not.

6.4 The Module System

Both Haskell and Erlang have a module system. When a refactoring allows the module interface to be changed, this refactoring may have an effect in not only the module where the refactoring is initiated, but also in those modules that import the current module. For example, renaming an exported identifier affects not only the module, `A` say, containing the identifier's definition, but also those modules importing `A`, directly or indirectly. How a number of differences between the Erlang and Haskell module systems affect the implementation of refactorings are discussed now.

In Haskell, an identifier exported by a module, `A` say, can be used by another module, `B` say, only if module B imports A either directly or indirectly; whereas in Erlang, a module can use functions (by remote function application) exported by another module, `B` say, without explicitly importing module B. From the point of view of program analysis, a module-level call graph of a Haskell program can be constructed by examining the import declarations of the modules in a program; whereas building a module-level call graph for an Erlang program needs the individual functions in each module of the program to be checked

Erlang does not allow transitive export of entities; that is, an Erlang module can only export functions which are declared in this module; whereas a Haskell module can export entities which are imported by this module. Therefore, an Erlang module, A say, does not propagate changes in its server modules (modules which are imported by A) to its client modules (modules which import A).

```
module M1 where

sq x = x ^ 2
---------------------------------------------
module M2(module M1, bar) where
import M1

bar x y = x + y
---------------------------------------------
module Main where
import M2

foo = x ^ 3
main x y = print $ foo x + bar x y
```

Figure 8. Adding a definition named `foo` to module `M1` would cause ambiguity in module `Main`

Omitting the export attribute in an Erlang module means that none of the functions defined in the module would be exported; whereas omitting the export list in a Haskell module means that *all* the names bound at the top level of the module would be exported.

Finally, in Haskell, an entity in the export list can be of the form `Module M`, which represents the set of all entities that are in scope with both an unqualified name "e" and a qualified name "M.e"; whereas, in Erlang, a function can only be exported by explicitly specifying its name and arity.

From the refactoring point of view, Haskell's export mechanism complicates the refactoring process when the module's export list is omitted or an entity like `Module M` is used in the export list. For example, when a new identifier is brought into scope in a module by a refactoring, the identifier could also be exported automatically by this module, and then further exported by other modules if this module is imported and exported by those modules. This is potentially dangerous as the new entity could cause name conflict/ambiguity in modules which import it either directly or indirectly, as shown in the example in Figure 8.

6.5 Static Typing vs. Dynamic Typing

Both Haskell and Erlang are typed programming languages, however Haskell features static typing whereas Erlang features dynamic typing.

Static typing could help the refactoring process (especially manual refactoring) by playing the role of testing during refactoring. Apart from that, type information is needed by some Haskell refactorings in order to succeed. For example, lifting a simple pattern binding (i.e. a pattern binding in which the pattern consists of only a single variable) to the top level may make an originally polymorphic definition monomorphic, with the result that the refactored program might fail to compile. This problem could be avoided by adding a proper type signature to the lifted pattern binding. Another example is that, when *generalising* a function definition which has a type signature declared, the type of the identified expression needs to be inferred and added to the type signature as the type of the function's first argument.

Erlang is a weakly typed programming language. For most Erlang refactorings, type information is not needed, but there are also some cases where type information can help. For example, type information has been used by Dialyzer [13], an Erlang tool that identifies software discrepancies, to detect redundant type tests.

6.6 Conclusions

We can make some general conclusions about Haskell and Erlang refactoring on the basis of this section.

Erlang is a smaller language than Haskell, and in its pure functional part, very straightforward to use. It does however have a number of irregularities in its static semantics, such as the fact that it is possible to have multiple defining occurrences of identifiers, and to nest scopes, despite the perception that there is no shadowing of identifiers in Erlang.

Erlang is also substantially complicated by its possibilities of reflection: function names, which are atoms. can be computed dynamically, and then called using the `apply` operator; similar remarks apply to modules. Thus, in principle it is impossible to give a complete analysis of the call structure of an Erlang system statically, and so the framing of side-conditions on refactorings which are both necessary and sufficient is impossible.

Two solutions to this present themselves. It is possible to frame *sufficient* conditions which prevent dynamic function invocation, hot code swap and so forth. Whilst these conditions can guarantee that behaviour is preserved, they will in practice be too stringent for the practical programmer. The other option is to *articulate* the conditions to the programmer, and to pass the responsibility of complying with them to him or her. This has the advantage of making explicit the conditions without over restricting the programmer through statically-checked conditions. It is, of course, possible to insert assertions into the transformed code to signal condition transgressions.

7 Implementation Considerations

Different techniques have been used in the implementation of HaRe and the Wrangler. HaRe is implemented

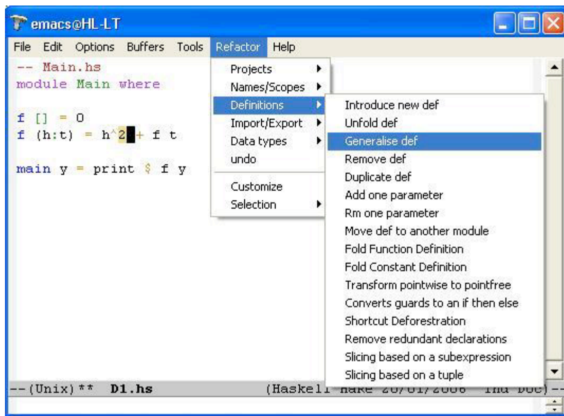

Figure 9. A snapshot of HaRe

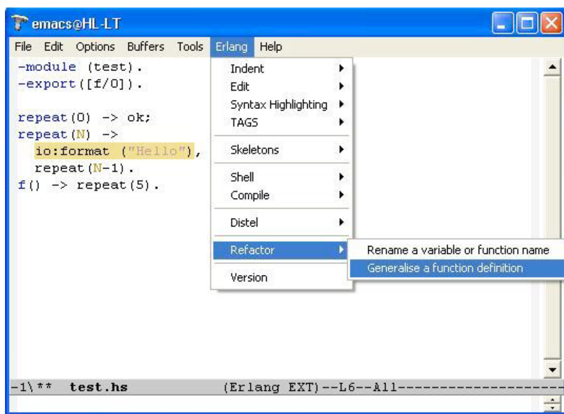

Figure 10. A snapshot of the Wrangler

in Haskell using the Programatica [9] frontend (including lexer, parser, module analysis, and type checker) for Haskell, and the Strafunski [6] library for generic AST traversals.

The Wrangler is implemented in Erlang using the Erlang Syntax Tools [11] library from the Erlang/OTP release and Distel [8] which is an extension of Emacs Lisp with Erlang-style processes and message passing and the Erlang distribution protocol. We have modified the Erlang Syntax Tools library to add the binding structure information to the AST.

HaRe is integrated with two most commonly used program editors for Haskell: (X)Emacs and Vim, while the Wrangler is integrated with Emacs, as most Erlang programmers use Emacs as the Erlang editor. Snapshots of HaRe and the Wrangler embedded in the Emacs environment are shown in Figure 9 and Figure 10 respectively. This section compares the implementation of the two refactorers.

7.1 The Implementation Language

HaRe is implemented in Haskell using the Programatica frontend [9], and Strafunski [6]. More detailed explanation of the implementation of HaRe can be found in [7].

When we started to prototype the Wrangler, we needed to decide which language we would use as the implementation language. After experimenting with the available Erlang frontends, we decided to commit to Erlang. The main reason is that almost all of available Erlang frontends are written in Erlang, and there is a library, i.e. Erlang Syntax Tools [11] from the Erlang release, which supports source-to-source program transformation. Apart from that, Distel [8] provides a very convenient way to integrated the refactoring tool with the Emacs editor. The other reason is that we feel that an Erlang refactorer implemented in the Erlang language might be more acceptable to the Erlang community; the same argument applies to the HaRe tool.

7.2 Analysis and Transformation

Program analysis and transformation involve frequent traversals of ASTs. Both HaRe and the Wrangler use an AST as the internal representation of the program under refactoring. However, unlike the Erlang ASTs produced by the Erlang Syntax Tools, in which all the non-leaf nodes have the same type, the Haskell ASTs are many-typed. While higher order function such as `map`, `fold` make traversals of Erlang ASTs less complicated, they do not help too much with traversing Haskell's complex, recursive, nested ASTs. The Haskell 98 abstract syntax contains a large number of mutual recursive algebraic data types, each being a sum of a large number of data constructors. A naïve traversal of a many typed AST could produce huge amount of boilerplate code, which is tedious to write and hard to maintain. To solve this problem, we used the Strafunski library to program AST traversal functions.

Strafunski is a library supporting strong-typed generic programming in application areas that involve term traversals over large abstract syntaxes. It is based on the notion of *functional strategy*. A functional strategy is a first-class generic function, which can be applied to arguments of *any* type, can exhibit *type-specific* behaviour, and can perform *generic* traversal to sub terms. Using Strafunski, we were able to write concise, type-safe, generic functions for AST traversals, in which only the strictly relevant constructors need to be mentioned.

7.3 Program Appearance Preservation

Program layout and comment preservation are issues when tool support for interactive refactoring is concerned. A real-world refactoring tool should preserve the original

program layout and comments as much as possible so that the program still looks familiar to the programmer after a refactoring has been applied. This requirement holds right across the programming spectrum as observed by J. R. Cordy in a keynote paper [2]. Unfortunately, most existing programming language frontends discard white space and comment information during the transformation from program source to the internal representation of the program, and most pretty-printing tools produce program source from ASTs by pretty-printing the layout and completely ignoring the original one. Together, this makes program appearance preservation a hard task.

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. After a refactoring, instead of pretty-printing the AST, we extract the source code from the transformed token stream. More details are presented in [7].

In the Wrangler, we again make use of the functionalities provided by Erlang Syntax Tools [11]. Erlang Syntax Tools provides functionalities for reading comment lines from Erlang source code, and for inserting comments as attachments on the AST at the correct places; and also the functionality for pretty printing of abstract Erlang syntax trees decorated with comments. The Wrangler therefore pretty-prints the transformed ASTs resulting from a refactoring.

8 Conclusions

In this paper, we have compared refactoring Haskell and Erlang programs from several aspects, including the useful refactorings for each language; the program analysis and transformation necessitated by typical refactorings; and the implementation of the refactoring tools. While both Haskell and Erlang benefit from the *referential transparency* property of functional programming languages, different language characteristics and programming idioms still affect the catalogue of useful refactorings, and the required program analysis and transformation of individual refactoring considerably.

The study begs the question of whether it is possible to design a generic refactoring tool which can be applied to several programming languages. The language-generic refactorer is an elusive beast. In theory, it should be possible to construct it, just as it should be possible to build semantically-driven compiler generator, which takes a description of a language and spits out a compiler, or at least its front end. What we have seen instead is that parser-generation technology is relatively common – though by no means universally used – but no production compiler is built automatically. As we have seen, a refactoring tool calls on all the front-end services of a compiler: static semantics, type checking, module analysis and so forth, and so all the practical obstacles to the full compiler generator carry over to the language-generic refactorer.

Another attack on the problem is API-oriented rather than semantic; we could see a generic system as providing a *toolkit* of language processing and transformation elements. A plethora of such systems exist, but in practice 'real' languages have sufficient peculiarities to make it difficult to embed their processing within such generic frameworks.

What is clear from our work is that *insights* are certainly transferable, and that implementing for Erlang a refactoring such as generalisation is made substantially easier with the experience of implementing it already for Haskell.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] J. R. Cordy. Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *International Workshop on Program Comprehension*, pages 196–206, 2003.

[3] FORSE. Formally-Based Tool Support for Erlang Development. http://www.cs.kent.ac.uk/projects/forse/.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] Huiqing Li and Simon Thompson and Claus Reinke. The Haskell Refactorer, HaRe, and its API. *Electr. Notes Theor. Comput. Sci.*, 141(4):29–34, 2005.

[6] R. Lämmel and J. Visser. Generic Programming with Strafunski. http://www.cs.vu.nl/Strafunski/, 2001.

[7] H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In J. Jeuring, editor, *ACM SIGPLAN Haskell Workshop, Uppsala, Sweden*, Aug. 2003.

[8] Luke Gorrie. Distel: Distributed Emacs Lisp (for Erlang).

[9] PacSoft. Programatica: Integrating Programming, Properties, and Validation. http://www.cse.ogi.edu/PacSoft/projects/programatica/.

[10] Refactor-fp. Refactoring Functional Programs. http://www.cs.kent.ac.uk/projects/refactor-fp/.

[11] Richard Carlsson. Erlang Syntax Tools. http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3/doc/html/.

[12] S.Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[13] T. Lindahl and K. Sagonas. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[14] S. Thompson. Refactoring Functional Programs. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, volume 3622 of *LNCS*, pages 331–357. Springer Verlag, September 2005.