A Domain-Specific Language for Scripting Refactorings in Erlang

Huiqing Li and Simon Thompson

School of Computing, University of Kent, UK
{H.Li,S.J.Thompson}@kent.ac.uk

Abstract. Refactoring is the process of changing the design of a program without changing its behaviour. Many refactoring tools have been developed for various programming languages; however, their support for composite refactorings – refactorings that are composed from a number of primitive refactorings – is limited. In particular, there is a lack of powerful and easy-to-use frameworks that allow users to script their own large-scale refactorings efficiently and effectively.

This paper introduces the domain-specific language framework of Wrangler – a refactoring and code inspection tool for Erlang programs – that allows users to script composite refactorings, test them and apply them on the fly. The composite refactorings are fully integrated into Wrangler and so can be previewed, applied and 'undone' interactively.

Keywords: analysis, API, DSL, Erlang, refactoring, transformation, Wrangler.

1 Introduction

Refactoring [1] is the process of changing the design of a program without changing what it does. A variety of refactoring tools have been developed to provide refactoring support for various programming languages, such as the Refactoring Browser for Smalltalk [2], IntelliJ Idea [3] for Java, ReSharper [3] for C#, VB.NET, Eclipse [4]'s refactoring support for C++, Java, and much more. For functional programming languages there is, for example, the HaRe [5] system for Haskell, and for Erlang the two systems Wrangler [6] and RefactorErl [7].

In their recent study on how programmers refactor in practice [8], Murphy-Hill et. al. 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 forty percent of refactorings performed using a tool occur in batches, they also claim that existing tools could be improved to support *batching* refactorings together.

Indeed, it is a common refactoring practice for a set of primitive refactorings to be applied in sequence in order to achieve a complex refactoring effect, or for a single primitive refactoring to be applied multiple times across a project to perform a large-scale refactoring. For example, a refactoring that extracts a sequence of expressions into a new function might be followed by refactorings

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 501–515, 2012.

that rename and re-order the parameters of the new function. This could be followed by 'folding' all instances of the new function body into applications of the new function, thus eliminating any clones of the original expression. As another example, in order to turn all 'camelCase' names into 'camel_case' format, a renaming refactoring will have to be applied to each candidate.

Although composite refactorings are applied very often in practice, tool support for composite refactorings lags behind. While some refactoring tools, such as the Eclipse LTK [9], expose an API for users to compose their own refactorings, these APIs are usually too low-level to be useful to the working programmer.

In this paper, we present a simple, but powerful, Domain Specific Language (DSL) based framework built into Wrangler, a user-extensible refactoring and code inspection tool for Erlang programs. The framework allows users to:

- script reusable composite refactorings from the existing refactorings in a declarative and program independent way;
- have fine control over the execution of each primitive refactoring step;
- control the propagation of failure during execution;
- generate refactoring commands in a lazy and dynamic way.

User-defined composite refactorings can be invoked from the *Refactor* menu in the IDE, and so benefit from features such as result preview, undo, etc.

Using the DSL allows us to write refactorings – such as the change of naming style discussed earlier – in a fraction of the time that would be required to do this by hand; we therefore make the cost of learning the DSL negligible in comparison to the benefits that accrue to the user. Moreover, once written these refactorings can be reused by the author and others.

Not only does the DSL make descriptions more compact, it also allows us to describe refactorings that are impossible to describe in advance. For example, suppose that a program element is renamed at some point in the operation; in the DSL we can refer to it by its old name rather than its new name, which may only be known once it has been input interactively during the transformation.

While this work is described in the context of Erlang. the underlying ideas and DSL design are equally applicable to other programming languages, and can be implemented in a variety of ways (e.g. reflection, meta-programming).

The rest of the paper is organised as follows. Section 2 gives an overview of Erlang, Wrangler and its template-based API. Section 3 formalises some concepts that we use in describing our work. In Section 4 we explain the rationale for our approach to designing the DSL, which we then describe in detail in Section 5. Examples are given in Section 6, and the implementation is discussed in Section 7. Sections 8 and 9 conclude after addressing related and future work.

The work reported here is supported by ProTest, EU FP7 project 215868.

2 Erlang, Wrangler and Its Template-Based API

Erlang is a strict, impure, dynamically typed functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance, and dynamic code loading.

```
-module (fact).
-export ([fac/1]).

fac(0) -> 1;
fac(N) when N > 0 ->
    N * fac(N-1).
```

Fig. 1. Factorial in Erlang

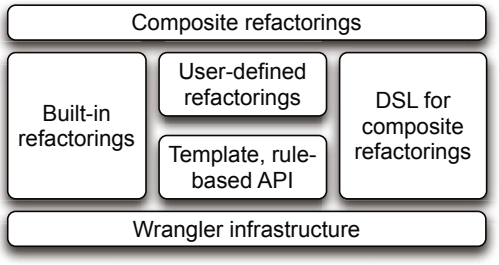

Fig. 2. The Wrangler Architecture

An Erlang program typically consists of a number of modules, each of which defines a collection of functions. Only functions exported explicitly through the `export` directive may be called from other modules; furthermore, a module may only export functions that are defined in the module itself.

Calls to functions defined in other modules generally qualify the function name with the module name: the function `foo` from the module `bar` is called as: `bar:foo(...)`. Figure 1 shows an Erlang module containing a 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 can represent entirely different functions computationally.

Wrangler [6], downloadable from `https://github.com/RefactoringTools`, is a tool that supports interactive refactoring and "code smell" inspection of Erlang programs, and is integrated with (X)Emacs and with Eclipse. Wrangler is itself implemented in Erlang. Abstract Syntax Trees (ASTs) expressed as Erlang data structures are used as the internal representation of Erlang programs. The AST representation is structured in such a way that all the AST nodes have a uniformed structure, and each node can be attached with various annotations, such as location, source-code comments, static-semantic information, etc.

One of the problems faced by refactoring tool developers is the fact that the number of refactorings that they are able to support through the tool is limited, whereas the number of potential refactorings is unbounded. With Wrangler, this problem is solved by providing a high-level *template- and rule-based API*, so that users can write refactorings that meet their own needs in a concise and intuitive way without having to understand the underlying AST representation and other implementation details. A similar strategy is used to solve the problem of describing composite refactorings, that is, a high-level DSL-based framework is provided to allow users to script their own composite refactorings. Wrangler's architecture is shown in Figure 2.

Wrangler's Template-based API [10] allows Erlang programmers to express program analysis and transformation in concrete Erlang syntax. In Wrangler, a code template is denoted by an Erlang macro `?T` whose only argument is the string representation of an Erlang code fragment that may contain meta-variables or

meta-atoms. A meta-variable is a placeholder for a syntax element in the program, or a sequence of syntax elements of the same kind; and a meta-atom is a place holder for a syntax element that can only be an atom, such as the function name part of a function definition.

Syntactically a meta-variable/atom is an Erlang variable/atom, ending with the character '`@`'. A meta-variable, or atom, ending with a single '`@`' represents a single language element, and matches a single subtree in the AST; a meta-variable ending with '`@@`' represents a list meta-variable that matches a sequence of elements of the same sort. For instance, the template

```
?T("erlang:spawn(Arg@@)")
```

matches the application of `spawn` to an arbitrary number of arguments, and `Args@@` is a placeholder for the sequence of arguments; whereas the template

```
?T("erlang:spawn(Args@@, Arg1@)")
```

only matches the applications of `spawn` to one or more arguments, where `Arg1@` is a placeholder for the last argument, and `Args@@` is the placeholder for the remaining leading arguments (if any).

Templates are matched at AST level, that is, the template's AST is pattern matched to the program's AST using structural pattern matching techniques. If the pattern matching succeeds, the meta-variables/atoms in the template are bound to AST subtrees, and the context and static semantic information attached to the AST subtrees matched can be retrieved through functions from the API suite provided by Wrangler.

The Erlang macro `?COLLECT` is defined to allow information collection from nodes that match the template specified and satisfies certain conditions. Calls to the macro `?COLLECT` have the format:

```
?COLLECT(Template, Collector, Cond)
```

in which `Template` is a template representation of the kind of code fragments of interest; `Cond` is an Erlang expression that evaluates to either `true` or `false`; and `Collector` is an Erlang expression which retrieves information from the current AST node. We call an application of the `?COLLECT` macro as a *collector*.

Information collection is typically accomplished by a tree-walking algorithm. In Wrangler, various AST traversal strategies have been defined, in the format of macros, to allow the walking of ASTs in different orders and/or for different purposes. A tree traversal strategy takes two arguments: the first is a list of collectors or transformation rules, and the second specifies the scope to which the analysis, or transformation, is applied to.

For example, the macro `?FULL_TD_TU` encapsulates a tree-walking algorithm that traverses the AST in a top-down order (`TD`), visits every node in the AST (`FULL`), and returns information collected during the traversal (`TU` for 'type unifying', as opposed to 'type preserving'). The code in Figure 3 shows how to collect all the application occurrences of function `lists:append/2` in an Erlang file. For each application occurrence, its source location is collected. `_This@` is a predefined meta-variable representing the current node that matches the template.

The template-based API can be used to retrieve information about a program during the scripting of composite refactorings, as will be shown in Section 6.

```
?FULL_TD_TU([?COLLECT(?T("lists:append(L1@, L2@)"),
                      api_refac:start_end_loc(_This@), true)], [File])
```

Fig. 3. Collect the application instances of lists:append/2

As was explained above, more details about the API can be found in [10]; in particular, it explains how the API can be used to define transformation rules which are also applied to ASTs by means of a tree-walking algorithm (as above).

3 Terminology

This section introduces terminology that we use in discussing our DSL. Particularly we explain what we mean by success and failure for a composite refactoring.

Definition 1. *A **precondition** is a predicate, possibly with parameters, over a program or a sub-program that returns either* `true` *or* `false`*.*

Definition 2. *A **transformation rule** maps one program into another.*

Definition 3. *A **primitive refactoring** is an elementary behaviour-preserving source-to-source program transformation that consists of a set of preconditions C, and a set of transformation rules T. When a primitive refactoring is applied to a program, all the preconditions are checked before the program is actually transformed by applying all the transformation rules. We say a primitive refactoring **fails** if the conjunction of the set of preconditions returns false; otherwise we say the primitive refactoring **succeeds**.*

Definition 4. ***Atomic composition** Given a sequence of refactorings $R_1, ..., R_n, n \geq 1$, the **atomic composition** of $R_1, ..., R_n$, denoted as $R_1 \circ R_2 \circ \cdots \circ R_n$, creates a new refactoring consisting of the sequential application of refactorings from R_1 to R_n.*

If any of the applications of $R_i, 1 \leq i \leq n$ fails, then the whole refactoring fails and the original program is returned unchanged. The composite refactoring succeeds if all the applications R_i for $1 \leq i \leq n$ succeeds, and the result program is the program returned after applying R_n.

Definition 5. ***Non-atomic composition** Given a sequence of refactorings $R_1, ..., R_n, n \geq 1$, the **non-atomic composition** of $R_1, ..., R_n$, denoted as $R_1 \diamond R_2 \diamond \cdots \diamond R_n$, creates a new refactoring consisting of the sequential application of refactorings from R_1 to R_n.*

If refactoring R_i fails, the execution of R_{i+1} continues, on the last succeeding application (or the original program if none has succeeded so far). A failed refactoring does not change the status of the program. The program returned by applying R_n is the final result of the application of the composite refactoring. As a convention, we say that a non-atomic composite refactoring always succeeds.

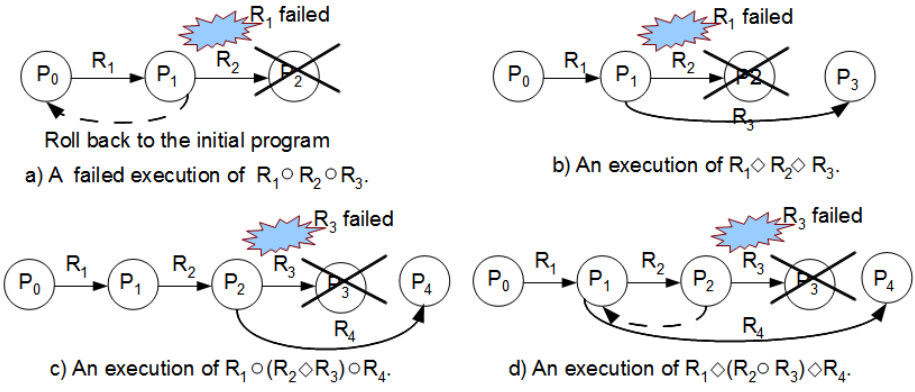

Fig. 4. Execution of composite refactorings

Figure 4 illustrates some execution scenarios of both atomic and non-atomic composite refactorings. As shown in c) and d), an atomic composite refactoring can be part of a non-atomic composite refactoring, and *vice versa*; this feature allows the tool to handle more complex refactoring scenarios.

In practice, the choice of atomic or non-atomic composition depends on the nature of the refactoring to be performed. Atomic composition is necessary if the failure of a constituent refactoring could lead to an inconsistent or incorrect program, whereas a non-atomic composition can be used when the failure of a constituent refactoring does not affect the consistency of the program, and the final program returned is still acceptable from the user's point of view.

For example, it is reasonable to make a non-atomic composition of a set of renaming refactorings that turn 'camelCase' function names into 'camel_case' format; even if one of these fails, perhaps because the new name is already used, the program still works as before. Moreover, the user can manually make the changes to the remaining 'camel_case' identifiers; if 90% of the work has been done by the script, 90% of user effort is correspondingly saved.

4 Rationale

Here we discuss the rationale for the design of the DSL. While it is possible to describe composite refactorings manually; that approach is limited:

- When the number of primitive refactoring steps involved is large, enumerating all the primitive refactoring commands could be tedious and error prone.
- The static composition of refactorings does not support generation of refactoring commands that are program-dependent or refactoring scenario dependent, or where a subsequent refactoring command is somehow dependent on the results of an earlier application.
- Some refactorings refer to program entities by source location instead of name, as this information may be extracted from cursor position in an editor or IDE, say. Tracking of locations is again tedious and error prone; furthermore, the

location of a program entity might be changed after a number of refactoring steps, and in that case locations become untrackable.
– Even though some refactorings refer to program entities by name (rather than location), the name of a program entity could also be changed after a number of refactoring steps, which makes the tracking of entity names hard or sometimes impossible, particularly when non-atomic composite refactorings are involved.

We resolve these problems in a number of ways:

– Each primitive refactoring has been extended with a *refactoring command generator* that can be used to generate refactoring commands in batch mode.
– A command generator can generate commands lazily, i.e., a refactoring command is generated only as it is to be applied, so we can make sure that the information gathered by the generator always reflects the latest status, including source locations, of the program under refactoring.
– Wrangler always allows a program entity to be referenced using its original name, as it performs name tracking behind the scenes.
– Finally, and most importantly, we provide a small domain-specific language (DSL) to allow composition of refactorings in a compact and intuitive way. The DSL allows users to have a fine control over the generation of refactoring commands and the interaction between the user and the refactoring engine so as to allow decision making during the execution of the composite refactoring.

Our work defines a small DSL, rather than a (fluent) API, since it supports a variety of ways of combining refactorings, including arbitrary nesting of refactoring descriptions within others, rather than offering a variety of parameters on a fixed set of API functions.

Existing approaches to composite refactoring tend to focus on the derivation of a combined precondition for a composite refactoring, so that the entire precondition of the composite refactoring can be checked on the initial program before performing any transformation [11,12]. The ostensible rationale for this is to give improved performance of the refactoring engine. However, given the usual way in which refactoring tools are used in practice – where the time to decide on the appropriate refactoring to apply will outweigh the execution time – we do not see that the efficiency gains that this approach might give are of primary importance to the user.

In contrast, our aim is to increase the *usability* and *applicability* of the refactoring tool, by expanding the way in which refactorings can be put together. Our work does not try to carry out precondition derivation, instead each primitive refactoring is executed in the same way as it is invoked individually, i.e., precondition checking followed by program transformation. While it may be less efficient when an atomic composite refactoring fails during the execution, it does have its advantages in expressibility.

5 A Framework for Scripting Composite Refactorings

In this section we give a detailed account of Wrangler's support for composite refactorings, treating each aspect of the DSL in turn.

5.1 Refactoring Command Generators

For each primitive refactoring we have introduced a corresponding *command generator* of the same name. The interface of a command generator is enriched in such a way that it accepts not only concrete values as a primitive refactoring does, but also structures that specify the constraints that a parameter should meet or structures that specify how the value for a parameter should be generated. In general, generators will have different type signatures, corresponding to the different signatures of their associated refactorings.

When applied to an Erlang program, a command generator searches the AST representation of the program for refactoring candidates according to the constraints on arguments. A command generator can also be instructed to run lazily or strictly; if applied strictly, it returns the complete list of primitive refactoring commands that can be generated in one go; otherwise, it returns a single refactoring command together with another command generator wrapped in a function closure, or an empty list if no more commands can be generated. Lazy refactoring command generation is especially useful when the primitive refactoring command refers some program entities by locations, or the effect of a previous refactoring could affect the refactorings that follow; on the other hand, strict refactoring command generation is useful for testing a command generator, as it gives the user an overall idea of the refactoring commands to be generated.

Each primitive refactoring command generated is a tuple in the format: {`refactoring, RefacName, Args`}, where `RefacName` is the name of the refactoring command, and `Args` is the list of the arguments for that refactoring command. A refactoring command generator is also syntactically represented as a three-element tuple, but with a different tag, in the format of {`refac_, RefacName, Args`}, where `RefacName` is the name of the command generator, and `Args` are the arguments that are specified by the user and supplied to the command generator. Both `refactoring` and `refac_` are Erlang atoms.

Taking the 'rename function' refactoring as an example, the type specification of the refactoring command is shown in Figure 5 (a), which should be clear enough to explain itself. The type specification of the command generator is given in Figure 5 (b). As it shows, a command generator accepts not only actual values, but also function closures that allow values to be generated by analysing the code to be refactored .

- The first parameter of the generator accepts either a file name, or a condition that a file (name) should satisfy to be refactored. In the latter case, Wrangler searches the program for files that meet the condition specified, and only those files are further analysed to generate values for the remaining parameters.
- The second parameter accepts either a function name tupled with its arity, or a condition that a function should meet in order to be refactored. In the latter case, every function in an Erlang file will be checked, and those functions that do not meet the condition are filtered out, and a primitive refactoring command is generated for each function that meets the condition.
- The third argument specifies how the new function name should be generated. It could be a fixed function name, a generator function that generates the

```
-spec rename_fun(File::filename(), FunNameArity::{atom(), integer()},
                 NewName::atom()) -> ok | {error, Reason::string()}.
```

(a) type spec of the 'rename function' refactoring.

```
-spec rename_fun(File::filename() | fun((filename()) -> boolean()),
                FunNameArity::{atom(), integer()}
                              | fun(({atom(),integer()}) -> boolean()),
                NewName::atom()
                  |{generator, fun(({filename(), {atom(), integer()}})
                                   -> atom())}
                     |{user_input,fun(({filename(), {atom(), integer()}})
                                   -> string())},
                 Lazy :: boolean())
          -> [{refactoring, rename_fun, Args::[term()]}] |
             {{refactoring, rename_fun, Args::[term()]}, function()}.
```

(b) type spec of the 'rename function' command generator.

```
   {refac_, rename_fun, [fun(_File)-> true end,
                  fun({FunName, _Arity}) -> is_camelCase(FunName) end,
                  {generator, fun({_File,{FunName,_Arity}}) ->
                                    camelCase_to_camel_case(FunName)
                                end}, false]}
```

(c) An instance of the 'rename function' command generator.

Fig. 5. Primitive refactoring command vs. refactoring command generator

new function based on the previous parameter values, or a name that will be supplied by the user before the execution of the refactoring, in which case the function closure is used to generate the prompt string that will be shown to the user when prompting for input.

– Finally, the last parameter allows the user to choose whether to generate the commands lazily or not.

The example shown in Figure 5 (c) illustrates the script for generating refactoring commands that rename all functions in a program whose name is in `camelCase` format to `camel_case` format. As the condition for the first parameter always returns true, every file in the program should be checked. The second argument checks if the function name is in `camelCase` format using the utility function `is_camelCase`, and a refactoring command is generated for each function whose name is in `camelCase` format. The new function name is generated by applying the utility function `camelCase_to_camel_case` to the old function name. In this example, we choose to generate the refactoring commands in a strict way.

For some command generators, it is also possible to specify the order in which the functions in an Erlang file are visited. By default, functions are visited as they occur in the file, but it is also possible for them to be visited according to the function callgraph in either top-down or bottom-up order.

$RefacName ::= rename\_fun \mid rename\_mod \mid rename\_var \mid new\_fun \mid gen\_fun \mid ...$
$PR ::= \{refactoring,\ RefacName,\ Args\}$
$CR ::= PR$
$\quad \mid \{interactive,\ Qualifier,\ [PRs]\}$
$\quad \mid \{repeat\_interactive,\ Qualifier,\ [PRs]\}$
$\quad \mid \{if\_then,\ \text{fun}() \rightarrow Cond \text{ end},\ CR\}$
$\quad \mid \{while,\ \text{fun}() \rightarrow Cond \text{ end},\ Qualifier,\ CR\}$
$\quad \mid \{Qualifier,\ [CRs]\}$
$PRs ::= PR \mid PRs,\ PR$
$CRs ::= CR \mid CRs,\ CR$
$Qualifier ::= atomic \mid non\_atomic$
$Args ::= ...A\ list\ of\ Erlang\ terms...$
$Cond ::= ...An\ Erlang\ expression\ that\ evaluates\ to\ a$ `boolean` $value...$

Fig. 6. The DSL for scripting composite refactorings

5.2 The Domain-Specific Language

To allow fine control over the generation of refactoring commands and the way a refactoring command to be run, we have defined a small language for scripting composite refactorings. The DSL, as shown in Figure 6, is defined in Erlang syntax, using tuples and atoms. In the definition, PR denotes a primitive refactoring, and CR denotes a composite refactoring. We explain the definition of CR in more detail now, and some examples are given in Section 6.

- A primitive refactoring is, by definition, an atomic composite refactoring.
- $\{interactive,\ Qualifier,\ [PRs]\}$ represents a list of primitive refactorings that to be executed in an interactive way, that is, before the execution of every primitive refactoring, Wrangler asks the user for confirmation that he/she really wants that refactoring to be applied. The confirmation question is generated automatically by Wrangler.
- $\{repeat\_interative,\ Qualifier,\ [PRs]\}$ also represents a list of primitive refactorings to be executed in an interactive way, but different from the previous one, it allows user to repeatedly apply one refactoring, with different parameters supplied, multiple times. The user-interaction is carried out before each run of a primitive refactoring.
- $\{if\_then,\ \text{fun}() \rightarrow Cond \text{ end},\ CR\}$ represents the conditional application of CR, i.e. CR is applied only if $Cond$, which is an Erlang expression, evaluates to `true`. We wrap $Cond$ in an Erlang function closure to delay its evaluation until it is needed.
- $\{while,\ \text{fun}() \rightarrow Cond \text{ end},\ Qualifier,\ CR\}$ allows CR, which is generated dynamically, to be continually applied until $Cond$ evaluates to `false`. $Qualifier$ specifies whether the refactoring is to be applied atomically or not.

– $\{Qualifier, [\ CRs\ ]\}$ represents the composition of a list of composite refactorings into a new composite refactoring, where the qualifier states whether the resulting refactoring is applied atomically or not.

5.3 Tracking of Entity Names

In a composite refactoring, it is possible that a refactoring needs to refer to a program entity that might have be renamed by previous refactoring steps. Tracking the change of names statically is problematic given the dynamic nature of a refactoring process.Wrangler allows users to refer to a program entity through its initial name, i.e. the name of the entity before the refactoring process is started. For this purpose, we have defined a macro `?current`. An entity name, tagged with its category, wrapped in a `?current` macro tells Wrangler that this entity might have been renamed, therefore Wrangler needs to search its renaming history, and replaces the macro application with the entity's latest name. If no renaming history can be found for that entity, its original name is used.

6 Examples

In this section, we demonstrate how the DSL, together with Wrangler's template-based API, can be used to script large-scale refactorings in practice. The examples are written in a deliberately verbose way for clarity. In practice, a collection of pre-defined macros can be used to write the script more concisely.

Example 1. Batch clone elimination Wrangler's similar code detection functionality [13] is able to detect code clones in an Erlang program, and help with the clone elimination process. For each set of code fragments that are clones to each other, Wrangler generates a function, named as `new_fun`, which represents the *least general common abstraction* of the set of clones; the application of this function can be then used to replace those cloned code fragments, therefore to eliminate code duplication. The general procedure to remove such a clone in Wrangler is to copy and paste the function `new_fun` into a module, then carry out a sequence of refactoring steps as follows:

– Rename the function to some name that reflects its meaning.
– Rename the variables if necessary, especially those variable names in the format of `NewVar`i, which are generated by the clone detector.
– Swap the order of parameters if necessary.
– Export this function if the cloned code fragments are from multiple modules.
– For each module that contains a cloned code fragment, apply the 'fold expression against function definition' refactoring to replace the cloned code fragments in that module with the application of the new function.

The above clone elimination process can be scripted as a composite refactoring as shown in Figure 7. The function takes four parameters as input:

– the name of the file to which the new function belongs,

```
1 batch_clone_removal(File, Fun, Arity, ModNames) ->
2   ModName = list_to_atom(filename:basename(File, ".erl")),
3   {atomic,
4     [{interactive, atomic,
5       {refac_, rename_fun, [File, {Fun, Arity},
7                             {user_input, fun(_)->"New name:" end},
8                             false]}},
9      {atomic, {refac_, rename_var,
10       [File, current_fa({ModName,Fun,Arity}),
11        fun(V) -> lists:prefix("NewVar", V) end,
12        {user_input,
13          fun({_, _MFA, V})->io_lib:format("Rename ~p to:", [V]) end},
14        true]}},
15     {repeat_interactive, atomic,
16       {refac_, swap_args, [File, current_fa({ModName, Fun, Arity}),
17                             {user_input, fun(_, _)->"Index 1: " end},
18                             {user_input, fun(_, _)->"Index 2: " end},
19                             false]}},
20     {if_then, [ModName] /= ModNames,
21       {atomic, {refac_, add_to_export,
22                 [File, current_fa({ModName, Fun, Arity}), false]}}},
23     {non_atomic, {refac, fold_expr,
24       [{file, fun(FileName)->M=filename:basename(FileName, ".erl"),
25                               lists:member(M, ModNames)
26               end}, ?current({mfa, {ModName, Fun, Arity}}),1, false]}}
27     ]}.
29 current_fa({Mod, Fun, Arity}) ->
30   {M, F, A} = ?current({mfa, {Mod, Fun, Arity}), {F, A}.
```

Fig. 7. Batch Clone Elimination

– the name of the new function and its arity,
– and the name of the modules that contain one or more of cloned code fragments, which is available from the clone report generated by the clone detector.

We explain the script in detail now.

– **Lines 4-8.** This lets the user decide whether to rename the function. The new name is provided by the user if the function is to be renamed.
– **Lines 9-14.** This section generates a sequence of 'rename variable' refactorings to form an atomic composite refactoring. Making this sequence of 'rename variable' refactorings atomic means that we expect all the renamings to succeed, however, in this particular scenario, it is also acceptable to make it non-atomic, which means that we allow a constituent renaming refactoring to fail, and if that happens the user could redo the renaming of that variable after the whole clone elimination process has been finished.
The second argument of the generator specifies the function to be searched. An utility function `current_fa`, as defined between lines 29-30, is used to ensure

```
tuple_args(Prog) ->
  Pars = ?STOP_TD_TU(
          [?COLLECT(?T("f@(As1@@, Line, Col, As2@@) when G@@ -> B@@."),
            {api_refac:fun_def_info(f@),length(As1@@)+1}, true)], Prog),
  {non_atomic, lists:append(
    [{refactoring,tuple_args,[MFA,Index,Index+1]}||{MFA, Index}<-Pars])}.
```

Fig. 8. Batch tupling of function arguments

the latest name is referenced. The function on line 11 gives the searching criterion for the variables to be renamed, and in this case it requires that all the variables with a name starting with "NewVar" should be renamed. New variable names are provided by the user as shown by the third argument. Refactoring commands are generated lazily, as indicated by the last argument, to ensure that the variables to be renamed are correctly identified.

- **Lines 15-19.** The code here allows re-ordering of function parameters. The user can choose to re-order as many times as necessary, or not at all.
- **Lines 20-22.** This generates a refactoring that adds the new function to the export of the module only if the clones are from multiple modules.
- **Lines 23-26.** Finally, this code generates a list of 'fold expression against function definition' refactoring commands, one for each module listed in `ModNames`. We allow these refactorings to be composed in a non_atomic way so that the refactoring process will continue if a refactoring fails for some reason.

Example 2. Batch tupling of function arguments The example in Figure 8 shows how Wrangler's template-based API can help to create composite refactorings. This example searches an Erlang program for single-clause function definitions whose parameters include `Line` and `Col` next to each other, and generates a 'tuple arguments' refactoring command for each candidate found to put `Line` and `Col` into a tuple. `Prog` specifies the scope of the project, i.e, the places to search for Erlang files.

7 Implementation

Wrangler has been extended with another layer to support scripted composite refactorings, and this includes a number of extensions as follows.

- *An interpreter of the DSL language.* The interpreter takes a composite refactoring script as input, and generates refactoring commands that to be executed by the refactoring engine. Only one refactoring command is passed to the refactorer engine a time. Depending on the result returned and the context, the interpreter could continue to generate another refactoring command or ask for a rollback of the program to a particular point if an atomic refactoring fails.
- *Support for rolling back* a program to the starting point of an atomic composition when it fails. This is an extension of Wrangler's original *undo* mechanism.
- *A command generator* for each primary refactoring as discussed in Section 5.1.

- *A mechanism for recording each primitive refactoring command executed.* Wrangler records each primitive refactoring command executed and the result returned during the execution of a composite refactoring. This information provides valuable insights into the refactoring commands generated/executed, as well as the reason of failure if some refactorings fail during the execution.
- *A generic composite refactoring behaviour.* A *behaviour* in Erlang is an application framework that is parameterized by a *callback* module. The behaviour solves the generic parts of the problem, while the callback module solves the specific parts. In this spirit, a behaviour, named *gen_composite_refac*, has been implemented especially for composite refactorings. Two callback functions are specified by the behaviour. To implement a composite refactoring, the user needs to create a callback module, implement and export the callback functions. Once the callback module is compiled, the refactoring can be invoked and tested from the IDE. The result can be previewed before being committed/aborted. A composite refactoring can also be undone.

8 Related Work

The idea of composite refactorings was proposed by Opdyke [14], and investigated by Roberts [15]. This work focused on the derivation of a composite refactoring's preconditions from the pre- and postconditions of its constituent refactorings.This is non-trivial because when performing refactorings $R_1, R_2, ...R_n$ sequentially, performing R_i may establish, or invalidate, the preconditions of $R_j, j > i$. Ó Cinnéide [12] extends Roberts' approach in various ways including static manual derivation of pre- and postconditions for a composite refactoring.

`ContTraCT` is an experimental refactoring editor for Java developed by G. Kniesel, et. al. [11]. It allows composition of larger refactorings from existing ones. The authors identify two basic composition operations: AND- and OR-sequence, which correspond to the atomic and non-atomic composition described in this paper. A formal model based on the notion of *backward transformation description* is used to derive the preconditions of an AND-sequence.

While the above approaches could potentially detect a composition that is deemed to fail earlier, they suffer the same limitations because of the static nature of the composition. Apart from that, the derivation of preconditions and postconditions requires preconditions to be atomic and canonical. In contrast, our approach might be less efficient when a composite refactoring fails because of the conflict of pre-conditions, but it allows dynamic and lazy generations of refactoring commands, dynamic generation of parameter values, conditional composition of refactorings, rich interaction between users and the refactoring engine, etc. Our approach is also less restrictive on the design of underlying refactoring engine.

The refactoring API – described in a companion paper [10] – uses the general style of 'strategic programming' in the style of Stratego [16]. More detailed references to related work in that area are to be found in [10].

9 Conclusions and Future Work

Support for scripting composite refactorings in a high-level way is one of those features that are desired by users, but not supported by most serious refactoring tools. In this paper, we present Wrangler's DSL and API[10] based approach for scripting composite refactorings. We believe that being able to allow users to compose their own refactorings is the crucial step towards solving the imbalance between the limited number of refactorings supported by a tool and the unlimited possible refactorings in practice.

Our future work goes in a number of directions. First, we would like to carry out case studies to see how the support for user-defined refactorings is perceived by users, and whether this changes the way they refactor their code; second, we will add more composite refactorings to Wrangler, but also make Wrangler a hub for users to contribute and share their refactoring scripts; and finally, we plan to explore the application of the approach to HaRe, which is a refactoring tool developed by the authors for Haskell programs.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
2. Roberts, D., Brant, J., Johnson, R.E.: A Refactoring Tool for Smalltalk. In: Theory and Practice of Object Systems, pp. 253–263 (1997)
3. JetBrains: JetBrains, `http://www.jetbrains.com`
4. Eclipse:: an open development platform, `http://www.eclipse.org/`
5. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer, HaRe, and its API. Electr. Notes Theor. Comput. Sci. 141(4), 29–34 (2005)
6. Li, H., et al.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada (2008)
7. Lövei, L., et al.: Introducing Records by Refactoring. In: Erlang 2007: Proceedings of the 2007 SIGPLAN Workshop on Erlang Workshop. ACM (2007)
8. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering 99 (2011)
9. Frenzel, L.: The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. Eclipse Magazine 5 (2006)
10. Li, H., Thompson, S.: A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK (2011)
11. Kniesel, G., Koch, H.: Static composition of refactorings. Sci. Comput. Program. 52 (August 2004)
12. Cinnéide, M.O.: Automated Application of Design Patterns: A Refactoring Approach. PhD thesis, University of Dublin, Trinity College (2000)
13. Li, H., Thompson, S.: Incremental Clone Detection and Elimination for Erlang Programs. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 356–370. Springer, Heidelberg (2011)
14. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, Univ. of Illinois (1992)
15. Roberts, D.B.: Practical Analysis for Refactoring. PhD thesis, Univ. of Illinois (1999)
16. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. 72 (2008)