A User-extensible Refactoring Tool for Erlang Programs

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 what it does. While it is possible to refactor a program by hand, tool support is considered invaluable as it allows large-scale refactorings to be performed easily. However, most refactoring tools are black boxes, supporting a fixed set of 'core' refactorings.
This paper reports the framework built into Wrangler – a refactoring and code inspection tool for Erlang programs – that allows users to define for themselves refactorings and code inspection functions that suit their needs. These are defined using a template- and rule-based program transformation and analysis API. User-defined refactorings are no "second-class citizens": like the existing ones supported by Wrangler, user-defined refactorings benefit from features such as results preview, layout preservation, selective refactoring, undo and so on.

Key words: Refactoring, Code inspection, Program analysis, Program transformation, API, Erlang behaviour, Wrangler.

1 Introduction

Refactoring [1] is the process of changing the design of a program without changing its behaviour. Refactoring tools support large-scale transformations, such as renaming a function or module, which require changes to all clients of the module or function across a project. Automation can guarantee not only that precondition checks are performed exhaustively, but also that the transformation itself respects the syntax and semantics of the language, thus making refactoring cheaper to perform (or undo) and also less error-prone.

While various refactoring tools have been developed to support different programming languages, most refactoring tools are black boxes, providing support for a fixed number of 'core' refactorings. New refactorings are not implemented for a number of reasons, including the complexity and cost of implementing a new refactoring, and the question of how generally applicable a new refactoring will be. This limits the scope of refactorings carried out by programmers in practice, and thus the opportunity to produce better code.

Wrangler is a refactoring and code inspection tool we have built for Erlang programs. Instead of being a black box, Wrangler provides a framework that

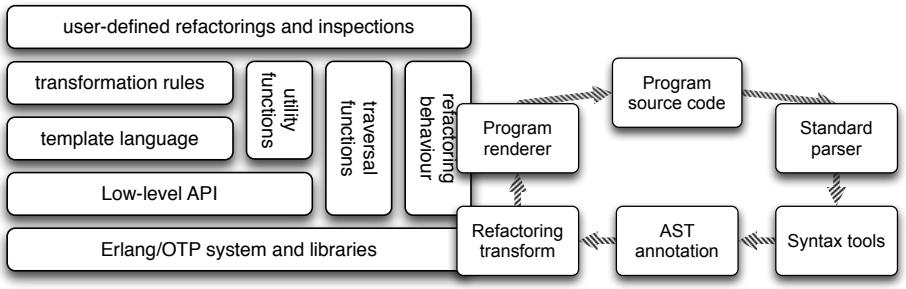

Fig. 1. The Wrangler API **Fig. 2.** The Wrangler workflow

allows users easily to define for themselves refactorings, code inspection functions and general program transformations that suit their needs.

The user-extensibility of Wrangler is achieved by introducing a new high-level layer – the *Wrangler API* – on top of the existing low-level API that provides direct access to the details of the internal representation of Erlang programs. What does the API need to do to support user-defined refactorings? A number of different services, as shown in Figure 1, together provide what is needed:

- A refactoring will transform portions of a program, and so we provide a *template language* to describe and match portions of programs. The template language extends the Erlang concrete syntax to include "meta-variables" that match portions of the program (in fact the program AST).
  For example, the template `?T("Mod@:f(Args@@)")` will match the application of a function named `f` in any module (the meta-variable `Mod@`) to any sequence of arguments (`Args@@`).
- Building on the template language we need a mechanism of *rules* to express the transformations from "old" to "new" program fragments, using the extended Erlang concrete syntax.
- While each rule describes how to transform the program "locally" – that is in one place – it is also necessary to describe how the rule is applied across the tree, and to do this we provide a set of *program traversal functions* that direct the application of the rules.
- The templates and traversal functions are also used to gather and collect the information needed to check the *pre-condition* of a refactoring before the transformation is applied.
- A collection of *utility* functions for various book-keeping tasks, such as abstract syntax tree traversal, retrieval of context information, mapping a textual selection within a program to its internal representation, etc.
- In order to integrate user-defined refactorings into interactive tools such as Wrangler in Emacs and Eclipse (ErlIDE) we describe a *workflow* which refactorings should follow. This is made concrete in an *interface* which each refactoring should implement; in Erlang this interface is known as a `behaviour`.

The main part of the paper is to introduce these features and to illustrate how they work in action through a series of examples.

User-defined refactorings that conform to the refactoring workflow (or Erlang `behaviour`) can be invoked from the *Refactor* menu in the IDE, Emacs say, and benefit from the existing features such as preview of refactoring results, layout preservation, selective refactoring, undo of refactorings, etc, for free. This allows a user to develop the refactoring in an iterative, test-driven, style.

A typical use of the API is to assist with *API migration functionality*. Most software will evolve during its lifetime, and this will often change the API of a library. However, API migration is generally not supported by refactoring tools due to the particularities of each individual API migration, but we can define these using the API. (It is also possible to *script* such refactorings using a domain-specific language for refactorings, reported elsewhere [2]).

The rest of the paper is organised thus. Sections 2 and 3 give brief overviews of Erlang and Wrangler. In Section 4, we introduce the template- and rule-based framework for analysis and transformation, and Section 5 explains the generic refactoring workflow. The work is evaluated in Section 6, and Section 7 covers related work. Section 8 makes some conclusions and discusses future work.

2 Erlang

Erlang [3] 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.

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. Calls to functions defined in other modules generally qualify the function name with the module name: the function `F` from the module `M` is called as: `M:F(...)`. Figure 3 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 can represent entirely different functions.

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

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

Fig. 3. Factorial in Erlang

In this paper we use various features of Erlang that support *meta-programming*. Many of the operations we define use Erlang macros, which provide source-level descriptions of transformations to be performed in advance of programs being compiled, and we use these as our principal implementation mechanism.

It is also possible to specify transformations over the parse trees directly. One of the options accepted by the Erlang compiler is {`parse_transform, Module`}. If this option is passed to the compiler, the user-defined function `Module:parse_transform/2` is called by the compiler and applied to the parsed code of the current module before the code is further processed. This is used by Wrangler to achieve template-based program transformation in concrete Erlang

syntax as discussed in Section 4. Finally, we rely on the `syntax_tools` library for a standard representation of Erlang abstract syntax in Erlang.

3 Wrangler

Wrangler [4] is an open source tool that supports interactive refactoring and "code smell" detection for Erlang programs. Wrangler is implemented in Erlang, and it is integrated with (X)Emacs as well as with Eclipse through the ErlIDE plugin. It supports a variety of structural refactorings, including process refactorings, as well as a set of "code smell" inspection operations and facilities to detect and eliminate duplicated code [5]. Wrangler is downloadable from `http://www.github.com/RefactoringTools`. An overview of the refactoring workflow in Wrangler is shown in Figure 2.

Wrangler uses an Abstract Syntax Tree (AST) as the internal representation of Erlang programs, as generated by applying the `syntax_tools` library [6] to the parse tree generated by the Erlang parser. The AST representation generated is designed so that all the AST nodes have a uniformed structure; building on this we extend nodes with various annotations such as location, static semantic information, etc.

A refactoring typically consists of two parts: *pre-condition* checking and *program transformation*. Both pre–condition checking and program transformation operate over the AST: during condition checking there is typically a phase in which information is gathered from across the tree and then collated, the transformations themselves are also typically accomplished by a tree walk.

Wrangler preserves the original layout of the program as much as possible. In order for users to be able to undertake refactoring in a speculative way as a part of their software development process, it is important to be able to undo any transformation. Wrangler also allows the user to preview the changes to be made by a refactoring, and the user would choose to commit the changes and finish the refactoring, or abort the changes leaving the original code unchanged.

Wrangler, as an interactive refactoring tool, allows the user to perform what we term *selective* refactorings. By this we mean refactorings that involve a single clause-local transformation, but may be applicable to multiple places across the project. An example of this kind of refactoring is to replace the use of `lists:map/2` with a list comprehension. Selective refactoring allows the user to choose which candidates to refactor, and which not to.

4 Template- and Rule-based Program Transformation

The mechanism we have defined for the user definition of refactorings are covered in this section. Before we cover the details we explore the rationale for what we have done in a little more detail.

A typical refactoring consists of two parts: program analysis and program transformation. In the original design of the system, both program analysis and

transformation require detailed knowledge of the AST representation and static semantic information, and this becomes one of the major barriers that prevent users from writing their own refactorings, or indeed general transformations. The *template* and *rule* based program analysis and transformation framework defined in this paper aims to allow Erlang users to express program analyses and transformations in Erlang concrete syntax, therefore eliminating the need to understand the details of the underlying AST representation. Refactorings are implemented in Erlang, which means a user does not need to learn a new language in order to write Erlang refactorings.

Under our approach program transformations are expressed as conditional transformation rules, each of which specifies the code template before the transformation, the code after transformation, and the condition for this rule to apply.

The task of program analysis is made easier in two ways. First, each AST node is annotated with rich context information as a result of the various static analysis techniques used by Wrangler, so that a user does not have to perform any complex static analysis. Second, a template-based information collection technique, together with an API suite, makes it straightforward to extract certain context information from a code fragment of interest.

Traversal of ASTs is needed in order to apply a transformation rule, or to collect some information from the program. Through the Wrangler API, a collection of pre-defined AST traversal strategies are provided, each of which serves a specific purpose. We look at each of these features in turn now.

4.1 Code Templates in Concrete Syntax

In Wrangler, a template is denoted by an Erlang macro `?T` whose only argument is the string representation of a code fragment that may contain meta-variables.

The template code fragment can represent a sequence of expressions, a function definition, an attribute, or a single function clause. As a convention, a template representing a function/attribute should always end with a full stop; whereas a single function clause must end with a semicolon, otherwise it will be interpreted as a function consisting of a single function clause by default.

A meta-variable is a placeholder for a syntax element in the program, or a sequence of syntax elements of the same kind. Templates syntactically are Erlang code, therefore the use of meta-variables in a template must not violate the syntactic correctness of the code fragment.

Syntactically a meta-variable is an Erlang variable ending with the character `'@'`. A variable not ending with `'@'` represents an object variable. Three kinds of meta-variables are supported:

– A meta-variable ending with a single `'@'` represents a single language element, and matches a single subtree in the AST. For example, the template

```
?T("M:F@(1, 2)")
```

represents a remote function call with a placeholder for the function name. In this template, variable `M` is an object variable, and only matches an AST node representing of a variable of the same name. On the other hand `F@` is

a meta-variable, and will match any node that represents the function name part of a fully-qualified [1] function call whose module name is `M`, and whose arguments are `1` and `2`.

- A meta-variable ending with `'@@'` represents a *list meta-variable*, which matches a sequence of elements of the same sort, e.g. a list of arguments of a function call, a sequence of expressions in a clause body, etc.
  For instance, the template `erlang:spawn(Arg@@)` matches the application of Erlang built-in function `spawn` to an arbitrary number of arguments (in Erlang, the spawn function can take 1, 2, 3, or 4 arguments); whereas the template `erlang:spawn(Args@@, Arg1@, Arg2@)` only matches the applications of function `spawn` to two or more arguments, where `Arg1@` and `Arg2@` are placeholders for the last two arguments respectively, and `Args@@` is the placeholder for the remaining leading arguments, if any.
- In Erlang, a function definition consists of one or more clauses, as does a complex expression (`case`, `if`, `receive`). In order to match against an arbitrary sequence of clauses, we introduce a special kind of meta-variable, which ends with `'@@@'`. A meta variable ending with `'@@@'` is mapped to a list, each element of which is a list of subtrees of the same kind. For example, a case expression with an arbitrary number of clauses can be matched thus:

  ```
  ?T("case Expr@ of Pats@@@ when Guards@@@ -> Body@@@").
  ```

  in which `Pats@@@` matches the collection of patterns from each clause of the case expression in the same order; `Body@@@` matches the collection of body expressions from each clause; and `Guard@@@` matches the collection of guards.

Meta-atoms. Certain syntax elements in Erlang, such as the function name part of a function definition, the record name/field in a record expression, etc, can only be an atom. In order to represent a placeholder for this kind of *atom-only* syntax elements, we introduce the notion of *meta-atom*. A meta-atom acts as a place holder for a single atom. Syntactically, a meta-atom is an Erlang atom ending with a single '@'. For example, with the use of meta-atom, an arbitrary function clause can be matched by:

```
?T("f@(Args@@)when Guard@@-> Body@@;")
```

where `f@` is a placeholder for the function name.

The same meta-variable or meta-atom name can be used multiple times in a template, in order to specify that two, or more, parts of the matching instance must be the same in order to match successfully. The only limitation is that all the location-related information annotated to the matching instance is no longer valid, and therefore removed from the matching instance bound to the meta-variable/atom.

4.2 Structural Pattern Matching

Erlang uses powerful pattern matching to bind variables to values. Similarly, structural pattern matching, with verification of some semantic conditions, is

[1] A fully-qualified function call in Erlang uses both the module and function names, separated by a colon.

one of the key operations used by Wrangler. Templates are matched at AST level, that is, the template's AST is pattern matched to the program's AST. If the pattern matching succeeds, the meta-variables and atoms in the template are bound to AST subtrees. A template consisting of single meta-variable will match any subtree of an AST.

4.3 Rule-based Conditional Transformation

A rule defines a basic step in the transformation of a program; it involves recognising a program fragment to transform and constructing a new program fragment to replace the old one. In Wrangler, a transformation rule is denoted by a macro `?RULE` with the format of:

```
?RULE(Template, NewCode, Cond),
```

where `Template` is a template representing the kind of code fragment to search for; `Cond` is an Erlang expression that evaluates to `true` or `false`; and `NewCode` is an Erlang expression that returns the new code fragment. Through a parse transform, all the meta-variables and atoms declared in `Template` are made visible to `NewCode` and `Cond`, and can therefore be referenced in defining them.

Applying a transformation rule to an AST entails pattern matching the AST representation of the `Template` against the AST. If successful, the `Cond` part of the rule is then executed. The condition is used to check that certain properties are satisfied by the bound instances of those meta-variables/atoms. The AST node is replaced only when the condition evaluation returns `true`.

The replacement AST node is constructed through the evaluation of `NewCode`. While the expression `NewCode` should evaluate to an AST, the user does not have to compose it manually; instead the general way is to create the string representation of the new code fragment, and use the macro `?TO_AST` to turn the string into its AST representation. The string representation of the replacement code could contain meta-variables and atoms, which are replaced with their bound instances after the string is parsed into AST. All the meta-variables and atoms bound in `Template` can be used by `NewCode`; furthermore, it is also possible for `NewCode` to define its own meta-variables to represent AST nodes.

For example, Figure 4 shows the rule to remove the `N`th argument from the application of function `M:F/A`, in which `M` represents the module in which the function is defined, `F` represents the name of the function, and `A` represents the arity of the function. `delete` is a normal Erlang function that deletes the `N`th element from the list `List`. `fun_def_info` is a Wrangler API function which takes an AST node as input, returns the `MFA` information of the node if it is associated with a function name, and otherwise returns `none`.

4.4 Support for Program Analysis

Program analysis plays a vital role in refactoring. Very often the program analysis process needs to collect some syntactic or semantic information from the AST. This task is supported by Wrangler in two ways. First, information derived from the program by Wrangler is attached to the AST as *annotations*: we

```
rule({M,F,A}, Nth) ->
  ?RULE(
    ?T("F@(Args@@)"),
    begin
      NewAs@@=delete(Nth, Args@@),
      ?TO_AST("F@(NewAs@@)")
    end,
    refac_api:fun_def_info(F@)
      == {M, F, A}).

delete(Nth, List) ->
  lists:sublist(List, Nth-1)++
    lists:nthtail(Nth, List).
```

Fig. 4. Remove the Nth argument

```
collect({M,F,A}, Nth)->
  ?COLLECT(
    ?T("f@(Pars@@)when G@@-> B@@;"),
    lists:nth(Nth, Pars@@),
    refac_api:fun_def_info(F@)
      == {M, F, A}).
```

Fig. 5. Collect the Nth parameter

```
?COLLECT(
  ?T("Body@@, V@=Expr@, V@"),
  {_File@,
    refac_api:start_end_loc(_This@)},
  refac_api:type(V@)==variable andalso
    [_]==refac_api:refs(V@)).
```

Fig. 6. Unnecessary match

provide functions to extract these annotations from the AST. Secondly, information available at different nodes can be *collected* together: our API provides a macro to support this collection.

Each node in the AST is annotated with rich context information as a result of the static analysis techniques being applied to the AST. Context information can be accessed through functions exported by the Wrangler API, including,

- Variables that are visible to the node, declared by the node, as well as variables that are used, but not declared, by the node. A variable is identified by the combination of variable name and the location of its binding occurrences (note that in Erlang a variable can have multiple binding occurrences);
- Source location information of the code fragment represented by the node;
- Syntax context information indicating the syntactic relation between the node itself and its parent node;
- Syntax category of the node, such as expression, pattern, operator, etc.
- For a node that refers to a function name, information regarding the definition of the function, e.g. the host module, the function name, arity, etc;.
- Binding and reference locations for a node representing a variable;
- The role played by an atom, i.e. function name, module name, process name, a literal, etc;
- For a node that represents a process identifier, information regarding how and where the process is spawned if this information can be inferred.

The macro `?COLLECT` is defined to allow information collection from nodes that match the template specified and satisfy certain conditions; it has the form:

`?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 returns the information extracted from the current AST node.

Applying an information collector to an AST node also entails pattern matching `Template`'s AST to the current AST node. If the pattern matching succeeds, the meta-variables/atoms in the template are bound to concrete AST nodes, and the `Cond` part of the collector is then executed. Information is extracted and returned if and only if `Cond` evaluates to `true`.

The example shown in Figure 5 can be used to collect the `N`th parameter of a function clause that defines function `M:F/A`. While the template specified in the collector only pattern matches a single function definition clause, when the collector is applied together with an AST traversal strategy, it will try to pattern match every function clause in the AST, as discussed in Section 4.5.

As another example, the macro application shown in Figure 6 collects those clause bodies with an unnecessary match expression at the end. This collector returns the location information of those clause bodies found. The condition part of this collector says that the meta-variable `V@` is bound to a variable, and this variable is only referenced once (not including the binding occurrence). Two special pre-defined meta-variables are used in this macro application. One is `_File@`, whose value is the file name of the source code to which the macro is applied to, or `none` is no such information is available; and the other one is `_This@`, whose value is the subtree that pattern matches the template.

4.5 AST Traversal Strategies

Each transformation rule or collection macro is "local" in that it will match particular sub-trees of an AST; in order for these operations to be applied across a complete AST it is necessary to use an AST traversal strategy. An AST traversal strategy walks through (some nodes of) the AST in certain order, and applies transformation rules to nodes that meet certain conditions or collects some information from each node visited when program analysis is concerned.

The complexity of the traversal strategy itself depends on the type of the AST nodes. For nodes that are homogenous, i.e. all nodes have the same type, traversal of ASTs can be easily achieved. The homogenous representation of AST nodes in Wrangler and the syntax type information stored in each node allow us to write generic functions that traverse into subtrees of the AST while treating most nodes in a uniformed way, but nodes with a specific type in a specific way.

A number of pre-defined AST traversal strategies are provided through the Wrangler API. Traversal strategies can be distinguished in three particular ways:

- The purpose of the traversal. There are 'type-unifying' traversals for collecting information; and 'type-preserving' traversals for AST transformation. The terms 'type-unifying' and 'type-preserving' are adopted from [7];
- The termination condition for the traversal. There are traversals that visit all the nodes of the AST, that are cut off below nodes where the conditional pattern matching succeeds, and traversals that stop after one successful conditional pattern matching;

– The order in which the AST nodes are visited. There are top-down traversals that first visit the parent node then the children nodes, and bottom-up traversals that first visit the children nodes then the parent node.

In Wrangler, a traversal strategy macro is named to reflect the three aspects above. For example, the traversal strategy `FULL_TD_TU` means that the AST is to be traversed in a top-down order, all the nodes in the AST will be visited, and the traversal will return the information collected.

A traversal strategy macro takes two arguments. The first is a collection of transformation rules or a collection of information collectors, and the second specifies the scope to which the transformation or analysis is to be applied.

When more than one transformation rule is supplied to a 'type-preserving' AST traversal strategy, the order of the rules matters: for each AST node, the traversal strategy starts with the first rule, and once a rule has been successfully applied, the rules following it will not be tried for this node. This feature makes sure that the transformation is deterministic. Unlike 'type-preserving' traversal strategies, a 'type-unifying' AST traversal strategy tries to apply every information collector to the AST node visited, and the union of the information collected is returned.

The example in Figure 7 shows part of the implementation of the *remove an argument* refactoring. In this example, two transformation rules are supplied to the traversal macro `?FULL_TD_TP`. The first removes the `Nth` parameter of the function definition; and the second removes the `Nth` argument from the application sites of function `M:F/A`. In the latter rule, instead of a template macro representing a function application as shown in Figure 4, we use the macro `?FUN_APPLY`, and turn to explaining that now.

```
remove_nth_arg(File, {M,F,A}, Nth) ->
 ?FULL_TD_TP([rule1({M,F,A},Nth),
              rule2({M,F,A},Nth)],[File]).

rule1({M,F,A}, Nth)->
 ?RULE(?T("f@(Pars@@) when G@@ -> Bs@@;"),
       begin
        NewPs@@=delete(Nth, Pars@@),
        ?TO_AST("f@(NewPs@@) when G@@->Bs@@;")
       end,
       refac_api:fun_def_info(f@)=={M,F,A}).

rule2({M,F,A}, Nth)->
 ?RULE(?FUN_APPLY(M,F,A),
       begin
        Args=api_refac:get_app_args(_This@),
        NewAs=delete(Ith, Args),
        api_refac:set_app_args(_This@, NewAs)
       end, true).
```

Fig. 7. Remove the Nth argument

4.6 Abstraction of Function Application

In Erlang, there are various ways to call a function. For example, to call a function `F` defined in module `M` with arguments `As`, one could write `F(As)` within module `M`, or `M:F(As)` in other modules, or `apply(M,F,[As])` as a meta-application, or `spawn(M,F,[As])` to invoke the function in another process, etc.

For a refactoring that changes the interface of a function definition, it is generally necessary for the refactoring to handle all these possible formats of function application. It would be a laborious task for a user to write a transformation rule for each kind of function application. To make this task easier, we have defined a special macro `?FUN_APPLY(M,F,A)` to represent a *meta-template* that can be used to match a function application in different formats. Together with the macro is a suite of *getter* and *setter* API functions that can be used to get, or set, a specific part of the function application. For instance, the function `get_app_fun/1` allows us to get the function name part of a function application; whereas the function `set_app_fun/2` allows to update the function name part of a function application. As shown in Figure 7, the use of `?FUN_APPLY` allows us to use only one rule to handle different function application scenarios.

5 Generic Refactoring Behaviour

While every refactoring has its own pre-condition analysis and transformation rules, there are some parts of the refactoring process that are generic to most refactorings, such as the generation and annotation of ASTs, the outputting of refactoring results, the collecting of change candidates, and the workflow of the refactoring process. We can use an Erlang *behavior* to capture this genericity.

A behaviour is an application framework that is parameterized by a *callback* module. The behaviour implements the generic parts of the problem, while the callback module implements the specific parts. A number of pre-defined behaviours are provided through Erlang OTP. In the same spirit, we have defined a behaviour especially for refactorings, and it is call *gen_refac*.

A user-interactive refactoring process generally follows this *workflow*: the user selects the focus of interest by either pointing the cursor to, or highlighting, a program entity, then invokes the refactoring command; if the refactoring needs initial inputs from the user, it then prompts the user for values; after these interactions, the refactor engine starts the pre-condition checking; the refactorer continues to carry out the program transformation if the pre-conditions are met, otherwise aborts the refactoring and returns the reason for failure.

For a refactoring that consists of a set of transformations e.g. generalisation of a function definition, it is necessary either to change all call sites or to change none; performing some only would lead to an inconsistent program. On the other hand, for a refactorings that only involves a clause-local transformation, but may be applicable to multiple places across the project, e.g. replacing the use of `lists:append/2` with the use of `++`, the user may want to have a chance to decide which candidates to refactor, and which not to. In the latter case, it would be ideal to allow the user to browse through all the candidate changes first, and then to decide which changes to commit.

With *gen_refac*, we aim to encapsulate those parts that are generic to all refactorings in the behaviour module, and let the user to handle the parts that are specific to the refactoring under consideration. Another advantage of having a refactoring behaviour is the ease of integration with the IDE. Since the inte-

```
-module(refac_replace_append). %%module name is also refactoring name.
-behaviour(gen_refac).
-export([input_par_prompts/0, select_focus/1,check_pre_cond/1,
         selective/0,transform/1]). %% Callback functions.

input_par_prompts() -> []. %% No user input is needed.

select_focus(_Args) -> {ok, none}. %% No focus selection is need.

check_pre_cond(_Args) -> ok.  %% No pre-condition.

selective() -> true. %% Allow selective refactoring.

transform(_Args=#args{search_paths=SearchPaths})->
   ?FULL_TD_TP([rule_replace_append()], SearchPaths).

rule_replace_append() ->
    ?RULE(?T("F@(L1@, L2@)"), ?TO_AST("L1@++L2@"),
          {lists,append,2} == refac_api:fun_def_info(F@)).
```

Fig. 8. Replace lists:append/2 with ++

gration only involves the IDE and the behaviour module, it is done by Wrangler; the developer of the callback module need not be concerned.

A number of *callback* functions are specified by *gen_refac*. To implement a refactoring, the user needs to implement the callback module, and export the callback functions. The callback functions specified by *gen_refac* include:

- `input_par_prompts()->[string()]`, which should return a lists of prompt strings, one for each input from the user;
- `select_focus(Args::#args{}) -> none |{ok, term()}`, which returns the initial selection, in the format of AST nodes or others, by the user if one is needed, otherwise `none`;
- `check_pre_condition(Args::#args{}) -> ok |{error, Reason}`, which returns `ok` if the pre_conditions are met, otherwise an error with the reason.
- `selective() -> true|false`, which should return `true` if the user is allowed to browse through and select the changes to be made.
- `transform(Args::#args())->{ok,[{filename(),ast()}]}|{error,Reason}`, which does the actual program transformation, and returns the new ASTs if successful, otherwise the reason for failure.

The predefined record `args` defines the data structure that is passed through, and can also be modified, by the different phases of the refactoring process.

As an example, the code in Fig 8 implements the refactoring that replaces the uses of `lists:append/2` with the use of `++` (to save space we omit the type specifications). This refactoring is straightforward, but it still shows the way that the *gen_refac* behaviour should be used.

When a user-defined refactoring goes wrong, accurate error messages are essential for debugging the implementation of refactoring. Various checkings have been built into Wrangler, including:

- Syntactic correctness checking of templates;
- Type correctness checking of macro arguments and the results returned by callback functions;
- Checking that makes sure it is syntactically legal to replace one AST node with another, and that no unbound variables are introduced, and so on.

6 Evaluation

A number of refactorings and code inspectors have been implemented using Wrangler's new framework, and they include *specialise a function definition*, *swap function arguments*, *delete a function argument*, *introduce/remove an import attribute*, a collection of clause-local transformations as described in [8], etc. Compared with the previous framework, Wrangler's new framework for composing refactorings has the following advantages:

- it allows the refactoring developer to think of the refactoring in terms of examples in concrete Erlang syntax instead of the AST, reducing the learning curve for writing refactorings;
- the code implementing a refactoring is considerably shorter and more readable;
- fewer comments are needed to document the implementation as the code explains itself very well;
- since the refactoring developer needs to concentrate only on the refactoring-specific parts, this speeds up the implementation process;
- finally, the open structure of Wrangler allows user to invoke their own refactorings from the *Refactor* menu without having to touch the Wrangler code, that is user defined refactorings can be stored separately from Wrangler code.

We have found a couple of minor limitations of the framework so far:

- Since the template macro `?T` only accepts certain types of code fragments, i.e. expression, function clause, function definition, and attribute, in order to template a code fragment that does not belong to these syntax categories, e.g. a generator used in a list comprehension, a larger code fragment which contains the code fragment of interest will have to be matched and deconstructed.
- The generation of replacement code in a transformation rule may lose some location information, therefore it is possible that the layout style of the replacement code produced by the refactorer is slightly different.

7 Related Work

Rule-based structural transformation is used by many other meta-programming paradigms, each of which has its own advantages and uses. TXL [9] is a generalised source-to-source translation system. A TXL program takes as input a

context-free grammar in BNF-like notation, and a set of transformation rules to be applied to inputs parsed using the grammar. TXL first parses inputs in the language described by the grammar, and then successively applies the transformation rules to the parsed input, producing as output the transformed source.

Stratego/XT [7] is a another language and toolset for program transformation. Stratego is a language for transformation of abstract syntax trees, and XT is a bundle of transformation tools that combines Stratego with tools for other aspects of program transformation, such as parsing, pretty-printing, etc.

Closely related to Stratego/XT is ASF+SDF [10]. ASF is a declarative language for specifying semantics of programming languages as algebraic rewrite rules, and SDF is a language for specifying syntax definition of languages. The ASF+SDF meta-environment supports program analysis, transformation, generation of interactive program environments, pretty-printer generation, etc.

All the above tools are powerful in the support of program transformation, but relatively weak in the support of program analysis. Of course, there are also some tools that are powerful in terms of program analysis, bot not so handy when it comes to program transformation. This makes these tools somehow less suitable for writing refactorings that involve complex program analysis and transformation, which is the case for most non-trivial refactorings.

As the successor of the ASF+SDF, RASCAL [11] is a language that aims to provide high-level integration of source code analysis and manipulation on the conceptual, syntactic, semantic and technical level. RASCAL is not released yet at the time of writing, and its effectiveness still needs to be assessed.

GenTL [12] is a logic-based meta-programming language that combines logic-based conditional transformations and concrete syntax patterns. In this paradigm, program analyses can be implemented as predicates and queries on the program representation and transformation can be achieved using the meta-programming features of Prolog for asserting and retracting logic clauses.

JunGL [13] is also a domain-specific language for implementing refactorings. It combines an imperative core with ML-like algebraic data types for representing syntax trees and functions defined by pattern matching with features for defining attributes on edges between AST nodes. The central mechanism for defining edges are path queries, which are a kind of regular expression for describing paths in the syntax tree. Instead of providing a high-level support for expressing transformations, JunGL relies on imperative modification of the AST.

In the context of functional programming languages, besides Wrangler, there are the HaRe tool for refactoring Haskell programs [14], the RefactorErl [15] and Tidier [8] tools for refactoring Erlang programs. Apart from Wrangler, all the other tools provide a set of 'core' refactorings. While HaRe provides an API to allow users to define their own refactorings, the API operates at the AST level, and knowledge of the definition of the abstract syntax is a necessity for writing refactorings. RefactorErl provides a query language to allow users to fetch various structural and static semantic information about the program, but it does not support users to define their own transformations. Tidier is a complete black-box refactoring tool supporting a collection of clause-local refactorings.

8 Conclusions and Future Work

In this paper, we have presented an API and behaviour exposed by Wrangler, in order to turn the black-box nature of Wrangler into an open system, so that users of Wrangler can make use of the powerful infrastructure provided by Wrangler to define their own refactorings, general program transformations, and code inspection functionalities. The new framework has been used by the authors of Wrangler to compose a number of refactorings.

The work reported here is for Erlang, and uses a number of features of the language well suited to meta-programming. However, we see that a similar approach would be possible for other programming languages, with different flavours depending on their particular paradigm and feature mix.

In the future, we would like to carry out case studies to see how the new framework is perceived and used by Wrangler users; we would also like to explore the application of the approach to HaRe, the refactoring tool for Haskell programs, in which case a type-aware pattern-matching is needed.

This research is supported by EU FP7 collaborative project ProTest (`http://www.protest-project.eu/`), grant number 215868.

References

1. M. Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
2. Li, H., Thompson, S.: A Domain-Specific Language for Scripting Refactoring In Erlang. Technical Report 5-11, School of Computing, Univ. of Kent, UK
3. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Inc. (2009)
4. Li, H., Thompson, S., Orosz, G., Töth, M.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, Canada. (September 2008)
5. Li, H., Thompson, S.: Incremental Code Clone Detection and Elimination for Erlang Programs. In: FASE 2011. (2011)
6. Richard Carlsson: Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3`
7. Bravenboer, M., Kalleberg, K.T., et al.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72**(1-2) (2008)
8. Sagonas, K.F., Avgerinos, T.: Automatic refactoring of erlang programs. In: Principles and Practice of Declarative Programming. (2009)
9. Cordy, J.R.: Source transformation, analysis and generation in txl. In: symposium on Partial evaluation and semantics-based program manipulation. (2006)
10. van den Brand, M., et al.: The ASF+SDF meta-environment: A component-based language development environment. In: Compiler Construction. Volume 44. (2001)
11. Klint, P., van der Storm, T., Vinju, J.J.: Rascal: A domain specific language for source code analysis and manipulation. In: SCAM 2009. (2009)
12. Appeltauer, M., Kniesel, G.: Towards concrete syntax patterns for logic-based transformation rules. In: Workshop on Rule-Based Programming. (2007)
13. Verbaere, M., et al.: Jungl: a scripting language for refactoring. In: ICSE'06. (2006)
14. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer, HaRe, and its API. Electr. Notes Theor. Comput. Sci. **141**(4) (2005)
15. Lövei, L., et al.: Introducing records by refactoring in erlang programs. In: Symposium on Programming Languages and Software Tools, SPLST 2007. (2007)