Automated API Migration in a User-Extensible Refactoring Tool for Erlang Programs

Huiqing Li
School of Computing
University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
School of Computing
University of Kent, UK
S.J.Thompson@kent.ac.uk

ABSTRACT

Wrangler is a refactoring and code inspection tool for Erlang programs. Apart from providing a set of built-in refactorings and code inspection functionalities, Wrangler allows users to define refactorings, code inspections, and general program transformations for themselves to suit their particular needs. These are defined using a template- and rule-based program transformation and analysis framework built into Wrangler.

This paper reports an extension to Wrangler's extension framework, supporting the automatic generation of API migration refactorings from a user-defined adapter module.

Categories and Subject Descriptors

D.2.3 [**SOFTWARE ENGINEERING**]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement

General Terms

Languages, Design

Keywords

Erlang, refactoring, API migration, Wrangler, software engineering, template, rewrite rule.

1. INTRODUCTION

Most software will evolve, and this will often change the API of a library, and such changes could potentially affect all client applications of the library, both locally and remotely. API migration is a process of refactoring, but API migrations are not generally supported by refactoring tools due to the specifics of each particular migration, and so the transformations required tend to be done manually by the maintainers of the client code, risking incorrectness.

This paper presents our approach to automating the implementation of API migration for Erlang. This work is built on top of Wrangler, a refactoring and code inspection tool for Erlang programs, but we note that the approach applies to other languages equally well. One of the features that distinguishes Wrangler from other refactoring tools is its user-extensibility, given by a template- and rule-based program analysis/transformation framework, allowing users to express their intentions using Erlang concrete syntax.

Our approach to automatic API migration works in this way: when an API function's interface is changed, the author of this API function implements an *adapter function*, defining calls to the old API in terms of the new. From this definition we automatically generate the refactoring that transforms the client code to use the new API. This refactoring can be supplied by the API writer to clients on library upgrade, allowing users to upgrade their code automatically.

As a design principle, we try to limit the scope of changes as much as possible, so that only the places where the 'old' API function is called are affected, and the remaining part of the code is unaffected. One could argue that the migration can be done by *unfolding* the function applications of the old API function using the adaptor function once it is defined. However, the code produced by this approach would be a far cry from what a user would have written. Instead, we aim to produce code that meets users' expectations.

The paper is organised thus: Sec. 2 introduces a running example, and Sec. 3 gives a brief overview of Wrangler and its template- and rule-based framework. Automated API migration in Wrangler is reported in Sec. 4, related work is covered in Sec. 5, and the paper is concluded in Sec. 6.

2. EXAMPLE: REGULAR EXPRESSIONS

As a running example we take the implementation of *regular expressions* in Erlang; the `regexp` library has been deprecated, and users are expected to use the `re` library, which has a somewhat different application programmer interface.

For instance, the function `match` from the `regexp` library is used to find the first longest match of regular expression `RegExp` in a `String`. If the match succeeds, the function returns a tuple `{match, Start, Length}` where `Start` is the starting position of the match, and `Length` is the length of the matching string; if the match fails it returns `nomatch`. Fig. 1 shows two examples that use the function; note that it would be possible to rewrite the `case` expressions in various different ways without changing their meaning.

Replacing uses of `match` in Fig. 1 with the corresponding functions in the `re` library gives Fig. 2. In particular, the replacement for `match` would be the `run` function with the option `global` set. The function `run` is different from `match` not only in the name, but also in inputs and outputs. The

ASE '12, September 3-7, 2012, Essen, Germany

```
secret_path(Path, [[NewDir] | Rest], Dir) ->
  case regexp:match(Path, NewDir) of
    {match, _Start, _Len} when Dir == to_be_found ->
      secret_path(Path, Rest, NewDir);
    {match, _Start, _Len} ->
      secret_path(Path, Rest, Dir);
    nomatch ->
      secret_path(Path, Rest, Dir)
  end.
                Code example (a)

document_name(Path) ->
  case regexp:match(Path,"[^/]*\$") of
    {match, Start, Len} ->
      string:substr(Path, Start, Len);
    nomatch -> "(none)"
  end.
               Code example (b)
```

Figure 1: Code examples using `regexp:match`

```
secret_path(Path, [[NewDir] | Rest], Dir) ->
  case re:run(Path, NewDir, [global]) of
    {match, _Match} when Dir == to_be_found ->
      secret_path(Path, Rest, NewDir);
    {match, _Match} ->
      secret_path(Path, Rest, Dir);
    nomatch ->
      secret_path(Path, Rest, Dir)
  end.
                Code example (a)

document_name(Path) ->
  case re:run(Path, "[^/]*\$", [global]) of
    {match, Match} ->
      {Start0, Len}=lists:last(lists:ukeysort(2,Match)),
      Start = Start0 + 1,
      string:substr(Path, Start, Len);
    nomatch -> "(none)"
  end.
               Code example (b)
```

Figure 2: Code after replacing 'match' with 'run'

domain of `match` is a proper subset of that of `run`, but the result of `run` upon successful matching however contains not only the longest match, but also *every* sub-pattern match, presented as a list of tuples of the form `{Start,Length}`. A further difference is that string indexing begins at 1 in the `regexp` library, but at 0 in `re`.

3. WRANGLER AND ITS API

Wrangler [1, 2, 3] is a tool that supports interactive refactoring and "code smell" detection for Erlang programs. Wrangler is integrated with Emacs and Eclipse (via ErlIDE). Wrangler uses annotated Abstract Syntax Trees (AAST) to represent Erlang programs, in which each AST node is annotated with static semantic information, location, etc. Wrangler's extensibility is achieved in a number of ways.

Implementation. Wrangler is implemented in Erlang, a language the users of Wrangler are familiar with.

Templates and Rules, as described in Section 3.1.

Infrastructure. A layer of commonly-used components which handles static analyses of Erlang programs, parsing programs into ASTs, AST traversals, rendering of new source code after a refactoring, support of undo, preview, etc.

Integration. To integrate user-defined refactorings into Wrangler, a *workflow* which refactorings should follow is defined as set of Erlang callbacks (called a `behaviour`). User-defined refactorings that implement the required callbacks can be invoked from the *Refactor* menu in Emacs or Eclipse.

3.1 Templates and Rules

The template- and rule-based API [1] allows programmers to express program analysis and transformation in Erlang concrete syntax. In Wrangler, a code template is indicated by the macro `?T` whose argument is the string representation of an Erlang code fragment that may contain meta-variables, which are placeholders for syntax element(s) in the program.

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

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

matches the applications of `spawn` function to one or more arguments, where `Arg1@` matches the last argument, and `Args@@` will match the remaining 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. 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 subtrees matched can be retrieved.

The template-based API is used not only to retrieve information about a program, but also to define program transformation rules. A rule defines a basic step in the transformation of a program, specifying a program fragment to transform and a new program fragment to replace the old one, and is denoted by a macro `?RULE` thus:

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

In the example `Template` is a template representing the code fragment to replace; `Cond` is a Boolean condition; and `NewCode` is an Erlang expression that returns the new code fragment as a string or an AST. The meta-variables declared in `Template` can be used in `NewCode` and `Cond`.

3.2 Meta-Rules

Various program transformations can be expressed using Wrangler's `RULE` macro, however the number and complexity of the rules one has to write could increase significantly when the transformation involves a sequence of sub-expressions or clauses, as in `case` expressions or function definitions. The main reason is that Erlang, like any other programming language, allows a programmer to make a number of – essentially arbitrary – decisions about the precise form of an expression to implement a particular intention.

For example, a `case` expression in Erlang can have multiple clauses, each made up of a pattern and a body: patterns are matched in turn, and the body of the first successful match is the result. While in some examples one clause has to come before another to preserve the semantics of the code, there are others in which the patterns are mutually exclusive, and the order in which they are written is arbitrary. A pattern may also have an optional boolean expression or

```
?META_RULE(
 ?T("case regexp:match(String@,RegExp@) of
      {match,Start@,Len@} when Guard1@@->Body1@@;
      nomatch when Guard2@@ -> Body2@@
     end"),
  "case re:run(String@,RegExp@,[global]) of
    {match,Match} when Guard1@@ ->
      {Start0,Len@} =
        lists:last(lists:ukeysort(2,Match)),
      Start@ = Start0 + 1,
      Body1@@;
    nomatch when Guard2@@ ->Body2@@
   end",
   api_refac:free_vars(Guard1@@) --
     (api_refac:bound_vars(Len@)++
         api_refac:bound_vars(Start@))==
      api_refac:free_vars(Guard1@@)).
```

Figure 3: An example meta-rule

```
match(String, RegExp) ->
 case re:run(String, RegExp, [global]) of
  {match, Match} ->
   {Start0,Len}=lists:last(lists:ukeysort(2, Match)),
   Start = Start0+1,
   {match, Start, Len};
  nomatch -> nomatch
 end.
```

Figure 4: Adapter function for `regexp:match/2`

guard which needs to be satisfied; the choice of whether or not to use a guard also adds to the decision of whether to use a `guard` expression also enlarges the design space.

To write transformation rules transforming case expressions that use `match` (such as those shown in Fig 1) to case expressions that use `re`, it is obviously going to be hard to capture all the possible scenarios using the `RULE` macro. To address this problem, we introduce the concept of a *meta rule*. A *meta rule* is also represented by a macro:

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

However, a number of syntactic constraints apply:

- `Template` and `NewCode` can only be a `case` or `try` expression. The clause patterns of `Template` should be mutually exclusive; as to `NewCode`, we only require that clause patterns together with clause guards ensure the mutual exclusiveness of expression clauses. Mutual exclusiveness guarantees that re-ordering of clauses does not change the semantics of the expression.
- If we index the top-level expression clauses of `Template` and `NewCode` as $T_1, ..., T_n$ and $N_1, ..., N_m$ respectively, then $m \leq n$ and there is a partial mapping from $T_1, ..., T_n$ to $N_1, ..., N_m$, that is, clause $N_{i(1 \leq i \leq m)}$ represents the transformation result of clause T_i, and for all $T_{i(i>m)}$, no code is generated; only the meta-variables declared in T_i can be referenced by N_i.

Unlike `RULE`, which pattern matches the `Template` code with object code just as they are, `META_RULE` does a more flexible pattern matching process. Take the `case` expression as an example, given a template `case` expression represented as :

`case` $Expr_1$ `of` T_1`;` T_2`, ...,` T_n `end`

and an object case expression represented as:

`case` $Expr_2$ `of` C_1`;` C_2`, ...` C_t `end`

where T_i,C_i are of the form: $Pattern_i$ `when` $Guard_i$`->`$Body_i$.

To pattern match the object case expression with the template case expression, $Expr_2$ is pattern matched with $Expr_1$, and each $C_{i(1<=i<=t)}$ is pattern matched with each $T_{j(1<=j<=n)}$. The pattern matching succeeds only if:

- $Expr_1$ and $Expr_2$ pattern match successfully;
- For each $C_{i(1<=i<=t)}$, either $Pattern_i$ is a catch-all pattern represented as an underscore '_' or an unused bound-variable, or *only one* expression clause from the template expression successfully pattern matches C_i. The 'only one' condition guarantees determinism when new object code is to be generated.

Upon a successful pattern matching and successful condition checking, the new object code is generated in this way:

- First the expression clauses, $N_1, ..., N_m$, in `NewCode` are replaced by expression clauses, $C'_1, ..., C'_t$, where C'_i is C_i if C_i is a catch-all clause, or N_j if C_i pattern matches successfully with T_j; in the latter case, the meta variables in N_j are replaced by their bound object code as a result of the pattern matching.
- Second, the 'new' object code is tidied up by removing declarations that are introduced by the transformation but remain unused and prefixing unused variables with underscore, etc, so that the transformation does not introduce warnings from the compiler.

As an example, the meta-rule in Fig 3 defines the transformation of case expressions using `regexp:match/2` to those using `re:run/3`. The condition of the rule says that the rule is applied only if none of the variables bound in the clause pattern of the template `case` expression is used by the guard expression of that clause. Applying the meta-rule to the code in Fig 1 generates the code in Fig 2. While the rule cannot refactor all the use cases of `match` into `run`, it handles a substantial proportion of them.

4. AUTOMATED API MIGRATION

Our approach to API migration is reported in the section.

4.1 The Adapter Module

An *adapter* function is a single-clause function that implements the 'old' API function using the 'new' API: the adapter function for `regexp:match/2` is shown in Fig 4.

A `case` expression is needed by the definition of the adapter function if the return value of the API function is affected by migration, and the return value is of a 'union' type. Within the `case` expression, each clause handles one type of the return value, and the clause body defines the 'old' value from the value returned by the 'new' API function. Guards can be used to ensure that the generated clauses do not overlap.

For an API migration that does not affect the return value of the function, a `case` expression is not needed, and the body of the adapter function could be just a function application of the 'new' function. A number of constraints should be satisfied by adapter functions:

- The definition should have only one clause, and the name/arity should be the same as the 'old' function.

- The parameters of the function should all be variables.
- If the function definition is a `case` expression, then the last expression of every clause body should be a simple expression that can be used as a pattern.

4.2 Generation of Transformation Rules

The rule generator takes an adapter function as input and generates a number of rules and meta-rules from it (at most 3, typically). For example, three rules will be generated from the adapter function defined in Fig 4:

A meta-rule with the template code as a `case` expression. The rule shown in Fig 3 is a slightly simplified version of the rule generated from the function in Fig 4. In this rule, the case expression argument, i.e. `regexp:match(String@, RegExp@)`, is derived from the name and parameters of the adapter function; a `case` expression clause is generated for every clause in the adapter function, and the clause pattern is inferred from the last expression of the corresponding `case` clause in the adapter function; the guard expression and clause body are meta-variables automatically generated.

The `NewCode` of the rule is derived from the body of the adapter function by removing the last expression of the each clause body, and adding the clause body/guard introduced in the `Template`. The `Cond` of the rule is a general condition that applies to most meta-rules, and is derived by analysing the patterns and guards of each clause in the template code.

A rule with the template code as a match expression. The left-hand side of the match expression is a place holder denoted by a meta-variable, and the right-hand side is `regexp:match(String@, RegExp@)`. i.e. a function application of the 'old' API function. The `NewCode` of the rule is the template match expression with its right-hand side replaced by the body of the adapter function; and `Cond` is `true`.

A rule with the template as a function application of the 'old' API function. In this case `NewCode` is the function body of the adapter function and `Cond` is `true`.

To avoid causing name capture/conflict when the rules are applied, all the new object code variable names used are fresh names automatically generated by Wrangler.

4.3 Applying an API Migration Refactoring

API migration refactorings are a special kind of refactorings, whose preconditions are always met. The way in which the refactoring rules are applied is also different from the way in which general rule-based refactorings are applied. As a matter of fact, the API migration process is a combination of rule application and refactoring. The following steps are followed when an API migration refactoring is applied.

Step 1: If there is a meta-rule, then it is applied first.

Step 2: If there is a rule with a match expression as the `Template`, first apply the *introduce a new variable* refactoring to every application of the 'old' API function that is a sub-expression of another expression (not including a `match` expression), so that a new `match` expression, which binds the function application to the newly introduced variable, is added before the inner-most enclosing expression statement of the function application; then apply the rule.

Step 3: The remaining rule, i.e. the rule whose template is a function application of the 'old' function, is then applied.

The purpose of the refactoring in step 2 is to avoid generation of expressions that are too complex. In order to keep the code generated as tidy as possible, apart from the refactoring step mentioned above, refactorings that get rid of unused expressions or variables are also applied after a rule has been applied, which is another difference between API migration refactorings and general refactorings.

5. RELATED WORK

The work most related to ours is reported by Lövei in [4], which aims to support automatic API migration for Erlang, but is different in two ways. First, in his approach, data flow analysis is used to trace the expressions affected by calls to the 'old' API function to determine the very last points in the data flow where the transformations can be applied, whereas we try to limit the scope of changes as much as possible; second, his approach requires the user to provide the migration rules; we just require the adapter module.

The survey [5] gives a taxonomy of different aspects of API migration: recommending replacements, deciding whether to make a change, actually making changes, and verifying the results. Our work fits the first and third categories. Notable in the works surveyed is the paper by Dig and his co-authors on the nature of API evolution [6].

Inference of refactorings in class upgrade is described in [7]; our work depends on the library writer to describe the API adapter, but then infers refactorings from this, to 'fold' the adapter into the client code. Bartolomei *et al* [8] describe patters for API migration by wrapping: our work shows how this can be extended by folding the wrapping into the code.

6. CONCLUSIONS AND FUTURE WORK

The work reported here is for Erlang, 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 will use the tool to generate API migration refactorings for API changes in the Erlang libraries; we also expect to carry out case studies to see how the approach is perceived and used by Wrangler users.

This research was supported by the EU FP7 collaborative projects ProTest (215868) and RELEASE (287510).

7. REFERENCES

[1] Li, H., Thompson, S.: A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK (2011)

[2] Li, H., Thompson, S.: Let's Make Refactoring Tools User-extensible! In Sommerlad, P., ed.: The Fifth ACM Workshop on Refactoring Tools. (June 2012)

[3] Li, H., Thompson, S.: A Domain-Specific Language for Scripting Refactoring In Erlang. In: FASE 2012. (2012)

[4] Lövei, L.: Automated Module Interface Upgrade. In: 8th ACM SIGPLAN workshop on Erlang. (2009)

[5] Nasser, V.H.: A Survey of Program Migration Methods. `http://pages.cpsc.ucalgary.ca/~vnasser/pub/comparisonofprogrammigration.pdf` (2010)

[6] Dig, D., Johnson, R.: How do APIs Evolve? A Story of Refactoring. J. Softw. Maint. Evol. **18**(2) (2006)

[7] Tansey, W., Tilevich, E.: Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications. In: OOPSLA '08, ACM (2008)

[8] Tonelli, T., Czarnecki, K., Lämmel, R.: Swing to SWT and back: Patterns for API migration by Wrapping. In: ICSM '10, IEEE Computer Society (2010) 1–10