Refactoring with Wrangler, updated

Data and process refactorings, and integration with Eclipse

Huiqing Li and Simon Thompson

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

György Orosz and Melinda Toth

Eötvös Loránd University, Budapest,
Computing Laboratory, University of Kent
{G.Orosz,M.Toth}@kent.ac.uk

Abstract

Wrangler is a refactoring tool for Erlang, implemented in Erlang. This paper reports the latest developments in Wrangler, which include improved user experience, the introduction of a number of data- and process-related refactorings, and also the implementation of an Eclipse plug-in which, together with Erlide, provides refactoring support for Erlang in Eclipse.

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; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) l anguages; Concurrent, distributed, and parallel languages; D.3.4 []: Processors

General Terms Languages, Design

Keywords Erlang, Wrangler, Eclipse, Erlide, refactoring, tuple, record, process, slicing

1. Introduction

Refactoring [9] is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (or remove) any bugs. While it is possible to refactor a program by hand, tool support is considered invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Refactoring tools [24] can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the application of the refactoring itself, thus making refactoring less painful and less error-prone.

Whilst the bulk of refactoring tools that have been developed have supported object-oriented programming, there is an increasing interest in refactoring tools for functional and concurrent languages. For Haskell there is HaRe [15, 16, 14], which is embedded in both the Emacs [4] and Vim [30] editors. A prototype of a refactoring tool for Clean is also available [29].

We have recently developed the Wrangler tool for refactoring Erlang programs [20, 17, 19, 21], and in [18] we and the team from Eötvös Loránd University, Budapest jointly reported work on our

system and their RefactorErl tool [23, 25]. In this paper we describe the latest developments in Wrangler, which include the introduction of a number of new refactorings, and also the implementation of an Eclipse plug-in which, together with Erlide, provides refactoring support for Erlang in Eclipse.

The rest of the paper is organized as follows. Section 2 gives a short overview of the Wrangler tool for refactoring Erlang programs. Section 3 reports several improvements to the Wrangler user experience. The next two sections describe the data-related refactorings: Section 4 the tupling of function arguments, and Section 5 the introduction of records. We move to discussing process-related refactorings in Section 6. The integration of Wrangler with Eclipse and Erlide is the subject of Section 7. Finally, we draw some conclusions and point to further work in Section 8.

2. Wrangler

Wrangler is a refactoring tool which supports interactive refactoring for Erlang programs. It is integrated with Emacs [4] and now also with Eclipse [7]. Snapshots of Wrangler embedded in Emacs and Eclipse are shown in Figure 1 and Figure 11. It uses Distel [11] to manage the communication between the refactoring tool and Emacs, and on the other hand the Eclipse integration uses RPC (Remote Procedure Call) to manage the communication.

Wrangler supports more than a dozen refactorings: *Rename variable/module/function*, *Generalise function definition*, *Move function definition to another module*, *Function extraction*, *Fold expression against function*, *Tuple function parameters*, *From tuple to record*, *Rename a process*, *Register a process*, *Add a tag to messages*, and *From function to process*. There are two functionalities for duplicated code detection: *expression search* within a single module and *duplicated code detection* across multiple modules.

2.1 Tool Structure

Every refactoring has two main parts: side-condition checking and transformation. In most cases the side-condition checking is more complex than the transformation itself, because it requires a lot of syntactic and semantic information to be collected and analysed, in order, for example, to ensure that the binding structure of the program is unaffected, or the way in which messages are passed between processes is unchanged. Figure 2 gives an overview the refactoring workflow in Wrangler.

Wrangler uses the standard Erlang parser, slightly modified to include more layout information, to parse an Erlang program into "parse trees", and the SyntaxTools [2] library to build the Abstract Syntax Tree (AST) representation of the program from the parse trees. The AST generated is then annotated with various kinds of syntactic and semantic information including locations, comments, syntax category information, binding structure information, hence becoming the term Annotated Abstract Syntax Tree (AAST). Both
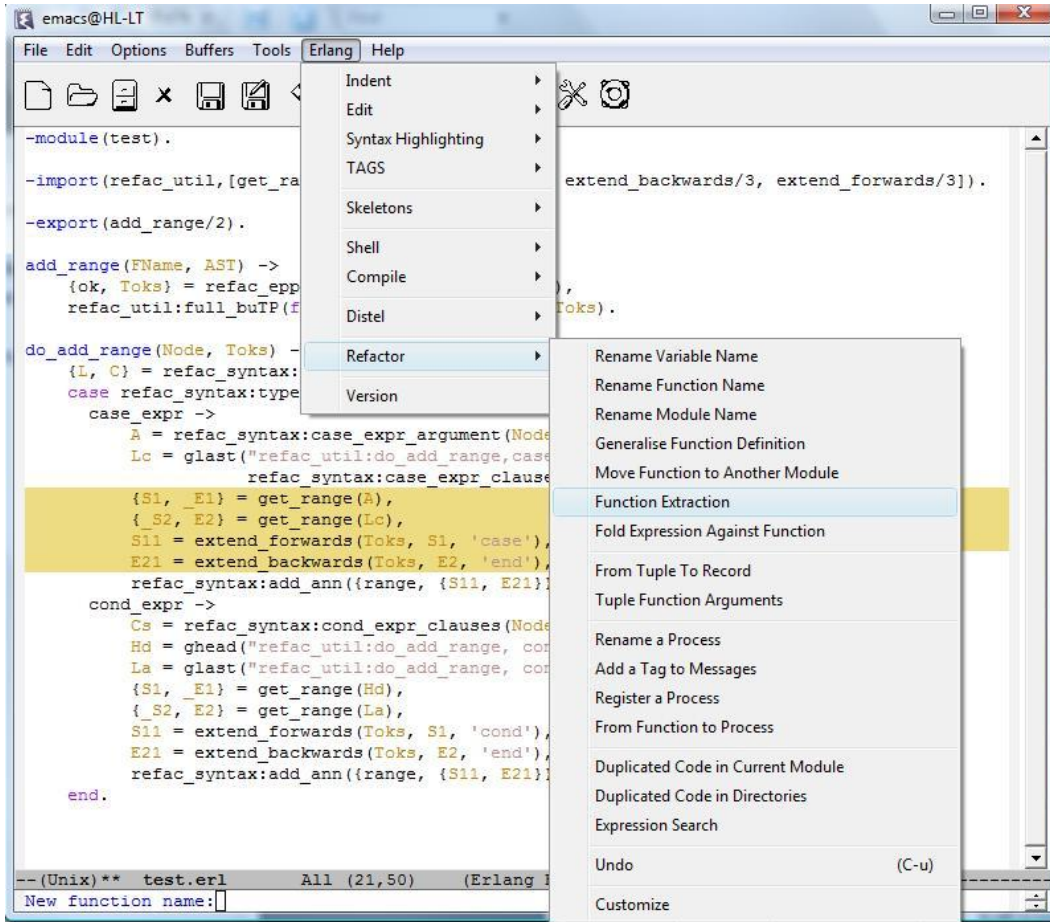
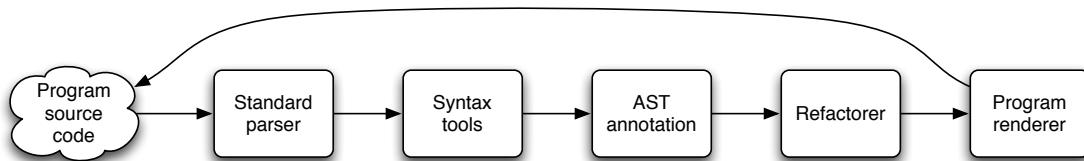**Figure 1.** A snapshot of Wrangler in Emacs

Figure 2. The Wrangler workflow

side-condition checking and program transformation operate over the AAST: during condition checking the conditions typically collate information gathered by walking the tree, and the transformations themselves are also typically accomplished by a tree-walking algorithm.

To perform a refactoring, the refactoring engine first gathers the necessary data and checks that all the side-conditions are satisfied, and then performs the necessary transformation if the previous check succeeds. Most of the refactorings need some user interaction when the refactoring is initiated (typically, a prompt for a new name), and/or during the refactoring process to allow the user to guide the refactoring process. All the refactorings supported by Wrangler are module-aware, supporting the refactoring of multiple-module projects.

Wrangler preserves the original layout of the program as much as possible, and functions/attributes that are not affected by a refactoring have the layout/comments unchanged after a refactoring. More about layout preservation is given in next section.

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. This can be done in Emacs, but if any edits have been performed after the last refactoring these will be lost; in the Eclipse embedding, the undo streams for edits and refactorings are fully integrated.

3. Improved User Experience

To better meet its users' expectations, the infrastructure of Wrangler has been modified in several ways, including improved program appearance preservation, support for refactoring code with syntax errors, and enhanced efficiency in the refactoring process. More details follow in the reminder of this section.

3.1 Program Appearance Preservation

By program appearance preservation, we mean that the refactored program should preserve the original program's layout and comment information as much as possible. Programmers would be reluctant to use a refactoring tool which reformats their code and makes it unrecognisable to them. Comment information is valuable for program understanding and long-term maintenance, therefore should never be discarded by the refactorer. Wrangler was designed to preserve comments, but not layout, from the very beginning.

Originally, we decided to use a pretty-printer to format the transformed program hoping that the layout produced would be acceptable by Erlang programmers; however, we soon discovered that this was not ideal. Our own refactoring experience suggested that sometimes the new layout produced could be so different from the original one that we would rather not to do the refactoring, hence we needed a better way to render the transformed program.

With the current implementation of Wrangler, program appearance preservation is achieved by making use of both token stream and AST. Erlang's standard token scanner discards both whitespace and comments from the source, however we have extended it to keep both. Location information, which is kept in both token stream and AST, is used to map the AST representation of an syntax phrase – such as a function, an attribute and so forth – to its token stream representation. After a refactoring, only those functions/attributes that are affected by the refactoring process are formatted by a pretty-printer, and all the other function/attributes are rendered by extracting the source from the token stream, therefore have their layout completely unchanged. The pretty-printer used by Wrangler respects the original layout, such as line width, of each function/attribute to be printed, and in most cases produces a layout very similar to the original layout of the function/attribute.

3.2 Refactoring Code with Syntax Errors

Wrangler has also been extended to accept Erlang programs that contain syntax errors or macro definitions that cannot be parsed by SyntaxTools. When the program under consideration has syntax errors or unparsable macros, functions/attributes to which these errors/macros belong are not refactored by the refactoring process, however warnings asking for manual inspection of those parts of the program will be given by Wrangler.

This feature was made possible in Wrangler by two facts. Firstly, the Erlang parser is self-recoverable, i.e., a function/attribute that does not parse does not stop the parser from parsing the code following it; secondly, location information kept in the AST and token stream allows us to extract the source code for those syntactically erroneous functions/attributes from the token stream, and put them back into the program during the program rendering process.

3.3 Efficiency Enhancement

While the program analysis and transformation needed by each refactoring may be different, all refactorings need to parse the program under consideration and annotate the AST produced, as shown in Figure 2. When refactoring a large project, a considerable amount of time could be spent on program parsing and annotation, and this could slow down the refactoring process. Therefore, it would be preferable if we could avoid the parsing and annotation process when it is possible, or parse and annotate the program when the refactoring engine is idle.

With Erlang as the implementation language of Wrangler, reusing of AAST is naturally achievable using Erlang processes, and indeed that is the approach we have adopted. With the latest implementation of Wrangler, a `gen_server` process, called *AST_server*, is dedicated to AST management. If an AAST is needed, the refactorer engine will ask *AST_server* for it. With *AST_server*, an Erlang module is parsed only when its AAST does not exist or is out-of-date. The refactoring engine also informs the *AST_server* when a module has been refactored, which will then update its AAST repository in the background. In a similar way, there is also a process in charge of maintaining the function/module callgraph in the background.

4. Tuple Function Parameters

The refactoring *Tuple Function Parameters* groups a number of consecutive arguments of a function into a tuple. This refactoring also modifies the arguments to the call sites of the function, and affects multiple modules if the function is exported, therefore has a global effect.

To apply this refactoring in Wrangler, the user first points the cursor to a function parameter or an application argument in the editor, then selects *Tuple Function Arguments* from the *Refactor* menu, after that the refactorer will prompt for the number of elements that are to form the new tuple.

Tuple Function Arguments has the following side-conditions:

- The indicated position in the editor must be a formal argument of a function definition or an application argument.

- The desired length of the tuple (*m*, say) must be valid. If the chosen parameter is the *n*-th element of the function arguments, and then *m+n-1* should not be larger than the arity of the function.

- The new function produced with a reduced arity should not conflict with existing functions.

- The function must not be an OTP callback function.

- As a design decision, we ask the user to initiate the refactoring from the module where the function is defined.

The example in Figure 3 illustrates an application of this refactoring which groups the first two parameters of function `f/3` into a tuple. Function `f/3` is exported by its defining module and used by another module, and in this case both the definition of `f/3` and its application in the other module `tup2` are changed. The `export` attribution is also affected by the refactoring.

```
―― Create tuple ――

-module(tup1).
-export([f/3]).

f(A, B, C) ->
   A+B+C.


-module(tup2).
-export([g/3]).

g(X, Y, Z) ->
   tup1:f(X, Y, Z).
```

→

```
―― Result ――

-module(tup1).
-export([f/2]).

f({A, B}, C) ->
   A+B+C.


-module(tup2).
-export([g/3]).

g(X, Y, Z) ->
   tup1:f({X,Y},Z).
```
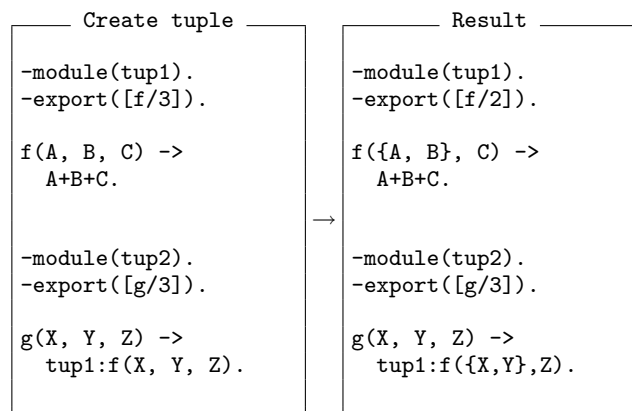
Figure 3. Tupling the first two arguments of the function `f`.

In the case that the function under consideration is used in an implicit *fun* application or a meta-function application, Wrangler will issue a warning message asking the user to check and modify manually if necessary.

5. Introduce Records

Erlang's principal data structuring mechanism is the *tuple*, which corresponds to the C structure, or indeed to tuples in other functional programming languages. The Erlang *record* allows tuple fields to be named, allowing programmers more flexibility in implementation by hiding some of the data representation. One example of this would be to allow a programmer to add a field to an existing record.

Thus, the process of turning a tuple into a record is a natural refactoring, which we call *From Tuple to Record*. This has been explored by the RefactorErl team [22], but to date remains a prototype in that system. We have chosen to take a bottom-up approach to implementing it in the work reported here.

Specifically we have chosen to implement the refactoring which transforms a tuple function parameter into a record expression. This refactoring modifies both the definition of the function and its application sites across the program. If the given record name does not exist, a new record definition is created by the refactorer.

In the remainder of this section we report the design and implementation of *From Tuple to Record*, and then explore ways in which this should be extended.

5.1 From Tuple to Record

To apply this refactoring in Wrangler, first mark a tuple in the editor, which should be a function parameter or an application argument, then select *From Tuple to Record* from the *Refactor* menu, and after that the refactorer will prompt for the record name and the record field names. As a design decision, the user should initiate the refactoring from the module where the function is defined.

A number of side-conditions are necessitated by this refactoring, and they are:

- The starting and ending positions of the selected text should delimit a tuple, which is a function parameter or an application argument.

- The given record name and field names should be atoms, and the record name should not have been used as a record name.

- The number of the field names given must be equal to the selected tuple size and must be distinct.

The example in Figure 4 shows the application of *From Tuple to Record* to the first argument of function `f/2`. A new record, named `rec`, with two fields has been created, and both the definition of `f/3` and its application in `g/1` have been changed. Since `f/2` is not exported by its defining module, this refactoring has a local effect; whereas the example in Figure 5 illustrates an application of this refactoring which affects multiple modules. In the latter example, both the definition of `f/3` and its application in the other module, `record2`, are affected. A record definition is created in both module `record1` and module `record2`. The resulting program could be further refactored by lifting the record definition into a `.hrl` file.

5.2 Types and the refactoring

The example in 6 illustrates refactoring a function which has more the one function clause, and can be applied to both tuples and lists. In this case the refactoring needs to analyze the function calls to the transformed function to decide whether an argument is a tuple (which will become a record) or not. In general this is not decidable, and so it will be necessary to add some run-time type checking (using `case` for example) to decide whether the argument

```
────── Create record expression ──────
-module(record).
-export([g/1]).

f({A, B}, C) ->
    A+B+C.

g(X) ->
    f({X, 2*X},3*X).
```

```
──────────────── Result ────────────────
-module(record).
-export([g/1]).
-record(rec,{first,second}).

f(#rec{first=A, second=B},C) ->
    A+B+C.

g(X) ->
   f(#rec{first=X,second=2*X},3*X).
```

Figure 4. An example of *From Tuple to Record* affecting a single module

```
────────────── Type example ──────────────
f({A, B}, C) ->
   A+B+C;
f([],C)-> 9.

h(X) ->
   Y = {X, X},
   f(Y, 5),
   S = []
   f(S, 3*X),
   Z = mod:app(X),
   f(Z, X).
```

Figure 6. Function with multiple clauses

is a record or not. This will clutter up the code, but serves as a warning to the possible user of a refactoring like this.

5.3 Replace tuple with record

In order to inform the next steps of our work, we have undertaken a case study of the Erlang Standard Library in order to discover the most used patterns of record usage. The three that we have discovered are

Replace tuple with record in a function body. Instead of accessing a tuple literally, we can name the record in the function argument and access it directly.

Using record update. If a tuple expression is a variant of another tuple expression, the former can be defined from the latter using record update syntax.

Record access. Access to components of a record can be given by an access expression, rather than by a pattern match of the whole record.

Used in combination, these transformations allow a user to hide the representation of a data type, thus giving a more abstract, and thus more flexible, interface to the data. It remains a research challenge to provide the appropriate interface to this collection of refactorings, so that a 'batch' application of them to a whole set of
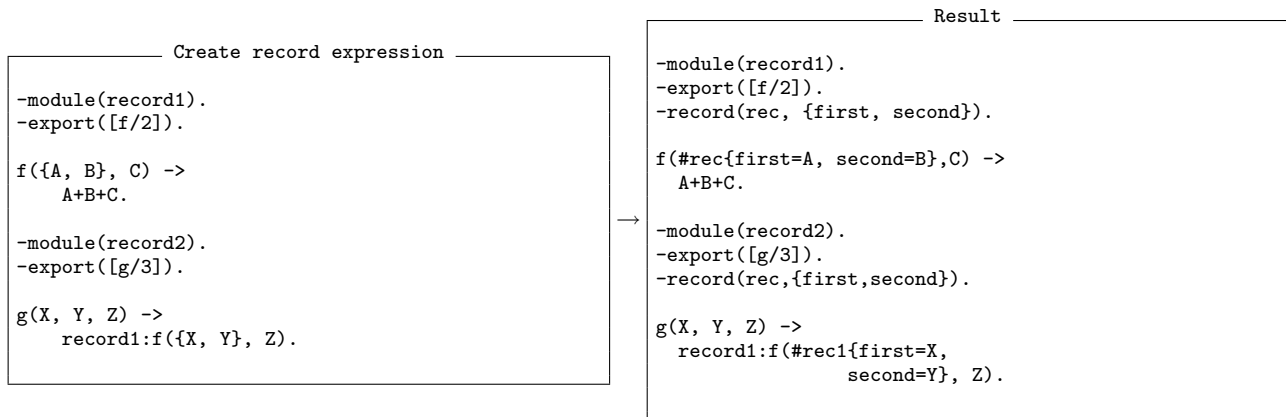
```
        ┌──── Create record expression ────┐    ┌──────────── Result ────────────┐
        │                                  │    │ -module(record1).              │
        │ -module(record1).                │    │ -export([f/2]).                │
        │ -export([f/2]).                  │    │ -record(rec, {first, second}). │
        │                                  │    │                                │
        │ f({A, B}, C) ->                  │    │ f(#rec{first=A, second=B},C) ->│
        │     A+B+C.                       │    │   A+B+C.                        │
        │                                  │ →  │                                │
        │ -module(record2).                │    │ -module(record2).              │
        │ -export([g/3]).                  │    │ -export([g/3]).                │
        │                                  │    │ -record(rec,{first,second}).   │
        │ g(X, Y, Z) ->                    │    │                                │
        │     record1:f({X, Y}, Z).        │    │ g(X, Y, Z) ->                  │
        │                                  │    │   record1:f(#rec1{first=X,     │
        │                                  │    │             second=Y}, Z).     │
        └──────────────────────────────────┘    └────────────────────────────────┘
```

Figure 5. An example of application of *From Tuple to Record* affecting multiple modules

functions which operate over a given (conceptual) data type can be devised.

6. Process-related Refactorings

Built-in support for lightweight processes is one of the strengths that distinguish Erlang from other programming languages. Erlang programs are made of lots of processes. These processes can communicate with each other by sending messages. In Erlang, programming with processes is easy, needs only three new primitives: **spawn**, **send (!)** and **receive**; however, undisciplined use of processes could make the program hard to understand and maintain. For example, some typical process-related bad code smells include

- Code for implementing a single process spans across multiple modules, or code for more than one kind of process exist in the same module.
- Use process and message passing when a function call can be used, or use sequential function calls to model parallel activity.
- Name of a registered process does not reflect its role or functionality.
- Send/receive untagged messages.
- Non tail-recursive functions, especially non tail-recursive servers.
- Register a process that only lives a short time, or not register a process that lives a long time
- Not use generic OTP libraries, such as the generic server, when doing so is more appropriate.

Most of the above bad code smells can be detected, and refactored out step by step manually. However, after having examined a few basic refactorings, such as *register a process*, *add a tag to messages*, we realised that the dynamic nature of the language and the implicitness of process and communication structure of an Erlang program present a challenge for tool support of automated process-related refactorings, or at least some of them.

For example, the refactoring *register a process* registers a process with a name provided by the user, and replaces the receiving process identifier in a send expression with the process name if the process identifier refers to, and only refers to, the selected process. An example application of this refactoring is shown in Figure 7. For this refactoring to be behaviour preserving, the following side-conditions are necessary:

- The process name provided by the user should be an atom, and should not have been used as a process name in the program under consideration.
- The selected process should not have been registered.
- Should multiple instances of the process exist during run time, they should not co-exist at the same time.
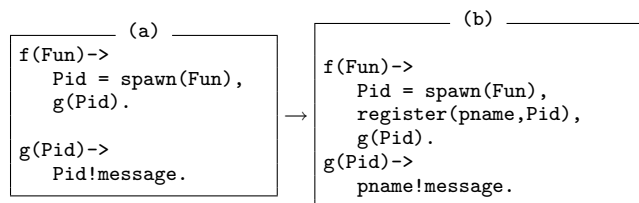
```
┌──────── (a) ────────┐      ┌──────── (b) ─────────┐
│ f(Fun)->            │      │ f(Fun)->             │
│     Pid = spawn(Fun),│     │     Pid = spawn(Fun),│
│   g(Pid).           │  →   │     register(pname,Pid),│
│                     │      │     g(Pid).          │
│ g(Pid)->            │      │ g(Pid)->             │
│   Pid!message.      │      │     pname!message.   │
└─────────────────────┘      └──────────────────────┘
```

Figure 7. Register a process

If all the side-conditions are met, we are then able to proceed with the transformation. However, when replacing a process identifier in a `send` expression with the process name, we must make sure that the process identifier *only* refers to the process selected. For instance, in the example shown in Figure 8, the `Pid` in expression `Pid!message` should not be replaced by `pname` because this Pid is associated with multiple process instances.

```
┌──────── (a) ────────┐      ┌──────── (b) ─────────┐
│ f(Fun)->            │      │ f(Fun)->             │
│     Pid = spawn(Fun),│     │     Pid = spawn(Fun),│
│     g(Pid),         │      │     register(pname,Pid),│
│     Pid1 = spawn(Fun),│    │     g(Pid),          │
│     g(Pid1).        │  →   │     Pid1 = spawn(Fun),│
│                     │      │     g(Pid1).         │
│ g(Pid)->            │      │ g(Pid)->             │
│   Pid!message.      │      │     Pid!message.     │
└─────────────────────┘      └──────────────────────┘
```
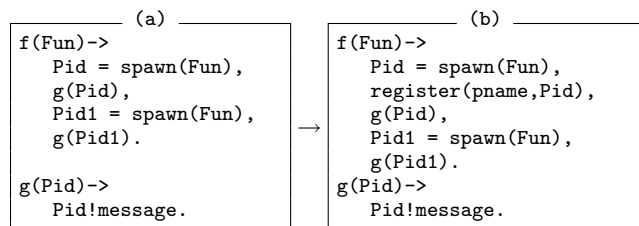
Figure 8. Register a process

Even though this refactoring is very basic, neither its side-condition analysis or its transformation is straightforward to carry out due to the dynamic feature of Erlang and the design of Erlang's process system. Next, we summarise the major challenges that we have encountered when process-oriented refactoring is concerned.

- Processes in an Erlang program are syntactically implicit. Unlike some other concurrency-oriented programming languages,

such as Pict [27] in which processes and channels are syntactically marked out, Erlang does not have a syntax category designed especially to identify processes. In an Erlang program, a process is created by the application of `spawn/1` or its variants. `spawn/1` itself is just an Erlang built-in function. For example, the expression

```
Pid = spawn(Fun)
```

creates a new concurrent process that evaluates `Fun`, and returns a `Pid` whose value identifies the process.

- Implicit connection between a process identifier and the process identified. The `spawn` expression above also reveals the fact that what identifies a process is not the name of the process identifier, but the actual value. Since a variable can take part in computation, or pass its value to other variables, it is possible that two or more process identifiers have the same value, therefore refer to the same process. Deciding whether two or more process identifiers refer to the same process statically needs data-flow analysis. Furthermore, as the Erlang type system only provides run-time rather than static type checking, even whether a variable stands for a process identifier or not is not always clear from the static view of the program.

  While it is possible to name a process using the function `register/2` provided by Erlang, it is not always desirable to do so especially if a process only lives a short time, and sometimes it is not possible to do so as pointed out by the side-conditions of *Register a process*.

- The process communication structure is implicit. Processes in an Erlang program communicate with each other by message passing. `Pid!Message` sends `Message` to the process identified by `Pid`, and returns the message itself; `receive...end` receives a message that has been sent to a process. Because of the indirect connection between a process identifier and the `send/receive` expressions of the identified process, trying to establish a connection between a `send` expression in one process and the corresponding `receive` expression in another process is difficult, not even to mention the mapping between particular messages sent/received. This is particularly obvious when the refactoring *Add a tag to messages* is concerned. This refactoring tries to add a tag to all the messages received (or sent) by a particular process, and obviously it needs to find out where these messages are sent from.

- Unlike functions or modules, a process in Erlang does not have a clear syntactically specified body or scope. Statically a process consists of the collection of functions that are reachable from the entry function/expression of this process. But, it is possible for multiple processes to share code, even `send/receive` expressions. Sharing of `send/receive` expressions makes it difficult to refactor messages sent/received, since it potentially affect all those processes sharing the code, as well as those processes that communicate with them.

- Process context dependent evaluations. Erlang is a language with side-effects. Some of the built-in functions provided by Erlang depend on the context of the current calling process. A particular example is the function `self/1`, which returns the process identifier of the calling process. Hence, care has to be taken if a refactoring changes the execution context of an expression. Examples of this kind of refactorings include *From function to process*, *From process to function*, *Spawn a new process to execute an expression* etc.

As mentioned before, Wrangler uses *annotated abstract syntax tree* (AAST) as the internal representation of Erlang programs. The annotation information includes binding information of variables and functions, syntax category, location, comment information, tokens, etc. Together with some fundamental functionalities for function call graph construction, module graph construction, side-effect analysis, etc, the existing infrastructure provides enough information to proceed with most refactorings regarding to the pure functional part of the language, but not with most process-related refactorings because of the challenges presented above.

To support process-related refactorings, we have extended our work in two aspects. Firstly, we have extended the existing AAST representation of Erlang programs with process information; secondly, we have exploited the use of slicing techniques to help the refactoring process. As a design strategy, Wrangler always try to extract as much necessary information as possible by static analysis, and minimise the amount of information needed from the user.

The remaining of this section is organised as follows. We first describe the annotation of AAST with process information, then discuss program slicing and its uses within the refactoring context. Finally, a summary of the process-related refactorings supported by the current implementation of Wrangler is given.

6.1 Annotate AST with Process Information

In an Erlang program, the only way to create a process is via the application of `spawn`, which creates a new concurrent process and returns a process identifier. But because process identifiers can be passed to other functions as parameters or returned values, or even passed to other processes by messages, sometimes it is not clear which process an identifier refers to. With this analysis, we aim to establish a static connection between a process identifier occurrence and the process identified. Due to the syntactic implicitness of Erlang processes, we use the `spawn` expression to represent the process created. In Wrangler, a particular `spawn` expression is identified by the combination of the spawn expression itself, the enclosing function of the spawn expression and the relative location of the spawn expression within the function. Location is needed to resolve the cases when two or more lexically the same spawn expressions occur in the same function.

As an example, given the sample code (a) in Figure 8, this analysis will annotate each occurrence of `Pid` in function `f/0` with

```
{pid, [{spawn(Fun), {mod, f, 1}, 1}}],
```

in which `pid` means the variable represents a process identifier, `spawn(Fun)` is the spawn expression that creates this identifier, `{mod,f,1}` refers to the enclosing function of the spawn expression, and the last integer `1` means that the spawn expression is the first spawn expression in this function. Here we assume that the name of the module to which the sample code belongs is `mod`. However, the occurrences of `Pid` in function `g/1` will be annotated with the following information because of the multiple application sites of this function:

```
{pid, [{spawn(Fun), {mod, f, 1}, 1},
       {spawn(Fun), {mod, f, 1}, 2}}].
```

With this kind of annotation, we are able to check whether two process identifiers refer to the same process or not by looking at the spawn expressions associated with them. The basic annotation algorithm used by Wrangler works as follows:

1. Construct the call graph for functions, and sort it topologically based on the dependencies between functions.

2. Within each function definition, annotate every occurrence of `spawn` application expression with process identifier information as illustrated above.

3. Analyze the call graph in a bottom-up order to propagate process information within each function definition through function application (when a function returns a process identifier), pattern matching and the binding structure of variables whenever it is possible. In the case that a function returns a process identifier, the return type of this function is also recorded.

4. Analyze the call graph in a top-down order to propagate process information from the call-sites to local function definitions.

5. Repeat from step 3 until a fix-point has been reached.

Apart from `spawn` expressions, process identifiers returned by other built-in functions, such as `self/1`, could also be annotated in a similar way.

So far, this algorithm does not handle complex pattern matching and message passing, therefore only partial process information is annotated into the AAST. However, methods have been taken to indicate whether the information annotated to a process identifier is complete or not.

User input is still needed when an undicidable situation occurs, but we try to reduce this kind of situations by the use of slicing techniques when it is possible.

6.2 Program Slicing

Apart from annotating AAST with process information, we have also exploited the use of program slicing techniques to reduce the number of uncertainties encountered by the refactoring engine by marking out the scope of the program that needs to be analysed or transformed.

The concept of program slicing was first introduced by Weiser. In [31], Weiser defines a program slice *S* as *a reduced executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P*. The slicing process generally starts for a *slicing criterion*, which represents the point in the code whose impact is to be observed with respect to the entire program. A backward slice contains all parts of a program that may have an effect on the criterion in question; by contrast, forward slices contain all parts of a program that may be affected by the selected criterion. Program slicing has its applications in many areas, such as debugging, code understanding, reverse engineering, program testing, etc. Program slicing itself could also be refactorings. For example, a function returning a tuple could be sliced into two functions, each of which returns an element of the tuple.

Within the context of refactoring Erlang programs, we have mainly exploited the use of static program slicing to reduce the scope of the program to be analysed, with the hope to reduce those undicidable cases for which Wrangler needs to ask for user's input or issue warning messages in order to proceed with the refactoring process. Both forward and backward inter-function slicing of Erlang programs have been implemented. In this paper we are not going into the details of the implementation, instead we focus on benefits of slicing during the refactoring process.

6.2.1 Forward slicing

Given an expression or a subset of the arguments of an Erlang function, Wrangler's forward slicer returns all parts of the program that may be affected by the value of the selected expression or arguments by employing data dependency analysis. The slicing algorithm operates cross function borders if the returned value of the function depends on the slicing criterion or any expression that depends on the slicing criterion is passed as a parameter to a function defined within the application in question. For instance, the example code (b) in Figure 9 shows the slicing result for the first `spawn(Fun)` expression in function `f/1`.

The major benefit of forward slicing is that it gives a clear scope of the program which might be dependent on the selected criterion,
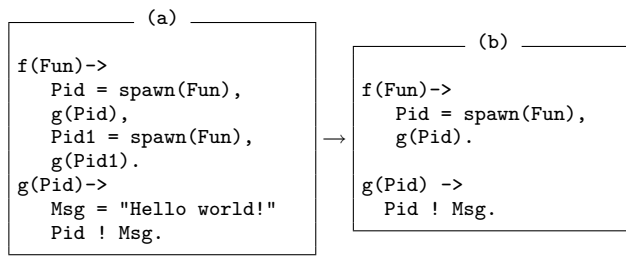
```
        ┌──────── (a) ────────┐
f(Fun)->
    Pid = spawn(Fun),                    ┌──────── (b) ────────┐
    g(Pid),                        f(Fun)->
    Pid1 = spawn(Fun),       →         Pid = spawn(Fun),
    g(Pid1).                           g(Pid).
g(Pid)->
    Msg = "Hello world!"           g(Pid) ->
    Pid ! Msg.                         Pid ! Msg.
```

Figure 9. Forward slicing

therefore a confined scope for program analysis if only the parts of the program that depend on the slicing criterion is necessary to be analysed. For example, to check whether a spawned process has been registered by other processes, we only need to check those registration expressions that belong to the slice produced by taking the spawn expression as slicing criterion. Reducing the analysis scope also reduces the number of undicidable situations encountered.

6.2.2 Backward slicing

In contrast to forward slicing, backward slicing uses a backward traversal of the data dependency flow from the point of interest given in the slicing criterion, and returns the parts of the program that could potentially affect the value of the selected expression. Depending on the applications of the computed slices, some will require that the returned slice is executable, while others only need the relevant expressions to be returned without checking whether those expressions form a syntactically well-formed program or not. With Wrangler, backward slicing has been used mainly with two scenarios. More details follow.

- Slice in order to evaluate. In some situations, it would help the refactoring process if Wrangler could know the possible values of a specific variable or expression. One approach is to use the functionalities provided by the module `erl_eval`, which defines an Erlang meta interpreter for expressions. For example, the function `erl_eval:exprs/2`, or its variants, can be used to evaluate a sequence of expressions in an abstract syntax representation. *First slice then evaluate* could ensure that only those expressions which could affect the value of the selected expression will be evaluated. More than that, in the case that the expression sequence to be evaluated depends on some formal parameters of the enclosing function, inter-function slicing provides more chances for the evaluation to be successful.

  For instance, with refactorings such as *rename a registered process*, *register a process*, Wrangler needs to know the process names that have already been used by the program, however this is not always straightforward when a process name can be dynamically composed as shown in the example code (a) in Figure 10. Taking the variable `ProcessName` from the expression `register(ProcessName, Pid)` as the slicing criterion, Wrangler's backward slicer will return the expression shown in part (b) in Figure 10. If there are multiple applications of the enclosing function of the slicing criterion, or functions that call this function either directly or indirectly, the slicer will return a list of expressions, each of which corresponds to a non-recursive call chain that leads to the function containing the slicing criterion. Note that it is not always the case that the produced slices can be evaluated, because of the lack of bindings for some functions for example, but again one strategy of Wrangler is to extract as much as information needed as possible.

```
                        ─── (a) ───
start() ->
  Prefix = "ch1",
  State = [1,2]
  start(Prefix, State).

 start(Prefix, State) ->
   ProcessName=list_to_atom(Prefix++"_proc"),
   Pid=spawn(ch1, init,[ProcessName, State]),
   register(ProcessName, Pid).
```

```
                        ─── (b) ───
fun(Prefix) ->
    ProcessName = list_to_atom(Prefix++"_proc"),
    ProcessName
end (begin Prefix = "ch1", Prefix end).
```

Figure 10. Backward slicing

- Like forward slicing, backward slicing can also be used to refine the scope of analysis. For example, taking a process identifier as the slicing criterion, backward could help to locate where the process is spawned, and even the initial function of the process identified.

The current slicing algorithms implemented in Wrangler do not handle process communication, and this aspect will be further investigated in the future.

6.3 Process-related refactorings supported by Wrangler

A number of process-related refactorings have been implemented using the enhanced infrastructure of Wrangler, and they are:

- *Register a process*, which register a process identifier with a user-provided name, and replaces the use of the process identifier in a `send` expression with the use of the process name whenever this is safe. Registering a process with a name allows any process in the system to communicate with the process without knowing its Pid.

- *From function to process*, which turns a function definition into a process, and all the calls to this function into communication with the new process. This refactoring provides potential for memorisation of the computed results and adding new functionalities.

- *Rename a registered process*, which renames a process' registered name to a user-provided new name. The main challenge of this refactoring is to detect whether an atom with the same name in the program presents a process name or not.

- *Add a tag to the messages sent/received by a process*, which adds a tag to all the messages received (or sent) by a process. This refactoring affects not only the process where the refactoring is initiated, but also the other processes which commutate with it. The refactoring does not distinguish individual messages received (or sent) by a process, therefore all the messages belonging to the processes involved will be added the same tag. The tags added can then be renamed manually by the user to distinguish different kinds of messages. While not ideal, this refactoring still help to mark out a clear scope that needs inspection.

7. Eclipse integration

There are some imitations to the way in which Wrangler is integrated into the Emacs editor, and so we have investigated integrating Wrangler in Integrated Development Environment (IDE).

In doing this we aimed to make as few changes to Wrangler as necessary, and to use it as a 'black box' to provide services to the IDE. On the other hand, this integration work provides a perspective on the design of Wrangler (and indeed Eclipse and its refactoring model) and we discuss this at the end of the section. Before that we describe the background to the work, and then give an overview of the integration work; full details of this work are given in the project report, [26].

7.1 Emacs

Emacs [4] is an highly configurable text editor with syntax highlight tool, debugger interface among many other features, but – as its name says *Editor MACroS* – it is just an editor with additional functionalities. What is more the fundamentals of the current version were originally written in 1984, when the developers of the tool, in a very understandable way, did not address refactoring support.

So the support provided by Emacs for various code transformation scenarios is not as good as it might be. To be more specific

- A typical refactoring will affect a complete project, rather than a single file. When integrating a refactoring tool with Emacs it therefore becomes necessary to define a notion of project, by, for instance, specifying a set of search paths.

- A number of refactorings – such as those which move a definition from one module to another, or those which rename a module – affect the way in which a project is built using 'make' or other systems. Changes made within the editor-embedded refactorer will not by default be reflected in the build infrastructure of the system.

- Emacs has a notion of 'undo', related to the editing operations; a refactoring tool will also provide a separate 'undo' operation; it is not at all clear how the two separate 'undo' operations can be put together.

Taken together these arguments against editor-embedded refactoring systems prompted us to investigate ways in which Wrangler could be integrated with an IDE.

7.2 Eclipse

The best developed open source IDE is Eclipse [7, 12], which *is an open source community whose projects are focused on building an extensible development platform, … for building, deploying and managing software across the entire software lifecycle. Many people know us … as a Java IDE but Eclipse is much more than [that]* [7]. In particular Eclipse has a plug-in architecture [5] which supports the integration of new functionality for Java and other languages. Plug-in distribution and update is provided by the Eclipse organisation.

For us, the most important thing is the refactoring support of Eclipse. It provides a very well documented refactoring API, the Eclipse Language Toolkit (LTK) [10], with fully support for integration into various aspects of the infrastructure of Eclipse, including

- the refactoring menu,

- refactoring previews, and,

- 'undo' and 'redo' support.

The LTK is described in more detail in Section 7.4 below, when we describe how Wrangler refactorings are integrated into Eclipse. Integration of this sort has already been developed for the Ruby language [6].

Eclipse is designed to be a universal tool platform and provides several extension points and APIs to extend it. The basis of Eclipse
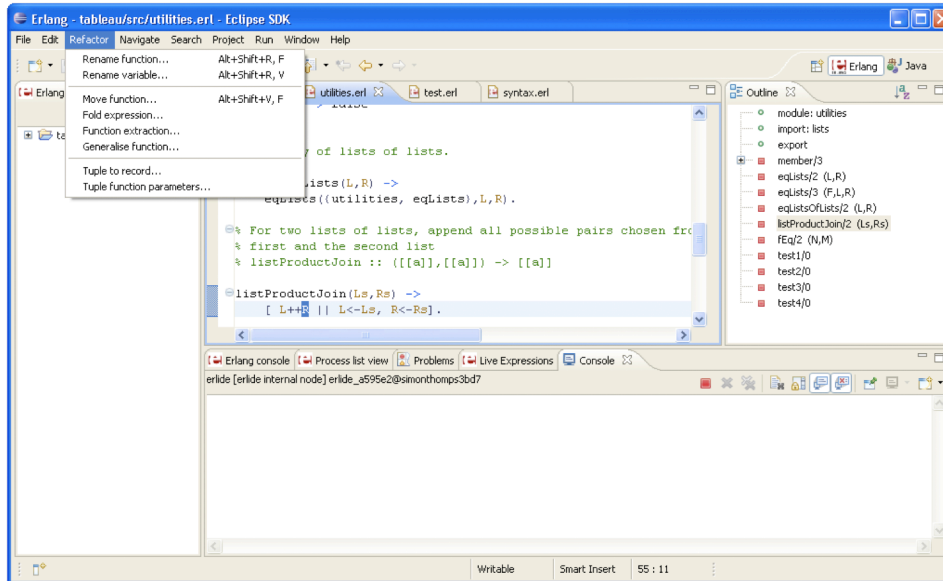
Figure 11. Wrangler in Erlide

is the kernel (or *runtime*), which loads plug-ins as needed. On top of this are four components

Workspace. The Workspace component handles the resources, including files, directories, projects, connections. Every modification of a resource is handled by the Workspace component; it also stores the history of each resource, letting the user undo or redo changes.

Workbench. The Workbench is the graphical interface next to the kernel. It is implemented in Eclipse's own Standard Widget Toolkit (SWT), giving OS native look-and-feel. It manages all the views, editors, and user actions as well. Of course it is also extensible using its extension points.

Team. This provides support for working with CVS / SVN repositories among other version management systems.

Help. This supports the definition and contribution of many kind of documentation.

Plug-ins can declare extension points, which can be used by others to extend its functionality in a controlled way. The Wrangler plug-in uses the following extension points:

org.eclipse.ui.editorActions: This allows plug-ins to add menus and toolbars to the workbench, when the selected editor type becomes active. In our case this was used to add the Refactor menu.

org.eclipse.ui.bindings: A binding is used to define relations between sets of conditions, commands and keybindings, and is used to create shortcuts for refactorings.

org.eclipse.ui.commands: This is used to create commands and command categories. A command is an abstract representation of a semantic behaviour; in our case it makes the connection between actions and bindings.

7.3 Erlide

The Erlide [8] plug-in provides an Eclipse-based development environment for Erlang, with features including a built-in console, automated build tool, syntax highlighting, code completion and de-

bugging support, outline and running processes view and live expression evaluation; see Figure 11. The Erlide backend is a Java interface for an Erlang node [3]. It provides thread safe RPCs (Remote Procedure Call) to each node, and each project is linked to a backend. This backend starts and stops when the project is opened or closed.

7.4 Integrating Refactorings using the LTK

The LTK provides a toolkit for integrating refactorings into Eclipse. This has a number of advantages, such as integrating them with the preview mechanism and the undo/redo mechanism, but it does provide a somewhat different workflow for refactorings than that assumed by Wrangler. An initial problem was that Wrangler is designed to modify source files, and we needed first to modify it so that it returns a new copy of the file. More fundamentally, the LTK workflow follows this pattern

1. The user initiates the refactoring.

2. An initial check is made of some of the preconditions.

3. User interactions (e.g. getting a new variable name).

4. According to the user input, another check is called; if no error occurs, the changes are calculated.

5. A preview dialog appears (optionally), then the calculated changes are applied if required.

while the Wrangler workflow is thus:

1. The user initiates the refactoring.

2. User interactions.

3. Applying the refactoring (within the Wrangler system)

    (a) checking conditions

    (b) calculating modifications

    (c) applying them to the AAST

    (d) writing them back to a new source file

Clearly, the Wrangler workflow will not allow the initial check (LTK 2) and so this stage becomes trivial, with user interactions

(LTK 3) preceding the call to Wrangler (LTK 4). This call will generate a new source file, from which a set of differences, calculated using an open source 'diff' tool, can be generated, as required by LTK 4. This 'diff' set forms the input for the final stage (LTK 5).

This correspondence gives a high-level overview of the way that a number of refactorings, such as *renaming functions and variables*, and *tupling of arguments*, can be integrated into Erlide and Eclipse. We next turn to some of the difficulties presented by the integration exercise.

7.5 Integration challenges

The model presented in the last section allows information to be gathered prior to any further processing, and this supports certain kinds of refactoring as discussed above. However, others require a more fine grained interaction. This includes *function generalization* and *folding expressions against function definitions*,which we discuss now.

In function generalization, a user selects a sub-expression of the function body, provides a new parameter name, and once this is done the user will be prompted by Wrangler for further confirmation in the case that the expression contains free variables or potentially causes a side-effect. This extra interaction is accommodated in the plug-in by means of Eclipse pop-up windows.

Folding instances of a function body into a call to that function will in general result in multiple instances of that body, and so multiple requests to the user for confirmation. In order to integrate this, it was necessary to change the Wrangler workflow for this refactoring, to return all the candidates in a single step, then to be iterated through within Erlide.

In both those cases, it was necessary to modify the refactoring to fit the LTK model of a refactoring. Some refactorings appear to go beyond the LTK model entirely. Any refactoring which modifies the files used by a system – such as renaming a module, or creating a new module by moving a definition to a non-existent module – cannot be accommodated in the LTK model.[1]

Other 'refactorings' – like clone detection – are not quite refactorings, and it would be artificial to include them in the LTK interface; we are currently investigating including them in a general 'search' interface.

7.6 Reflections on Wrangler

The LTK workflow presented in Section 7.4 suggests that the architecture of Wrangler might be modified to fit more tightly into Eclipse. In particular, it would be possible to refactor the preconditions of refactorings into two parts.

- The first part could be checked independently of the user input: in the example of 'rename function' this might include checking that the current position of the cursor is on a function identifier.

- The second part will use the user input – in our example the new name for the function – and check that, for instance, this name is not already used in the module, or imported from another module.

As we have noted earlier, the output of Wrangler after a refactoring is a new file, from which we calculate a 'diff' set; it would be possible to modify Wrangler to produce a 'diff' set directly. We aim to investigate these modifications in the months to come, and to continue our overall project to integrate Wrangler as tightly as possible into Eclipse and Erlide.

---

[1] The Eclipse Java refactoring systems allows these file changes, but note that it was implemented prior to the definition of the LTK.

8. Conclusions and future work

It is clear that as we look at more advanced refactorings – such as those involving wholesale transformation of data representations, or others which address inter-process communication – then more complicated analyses are required. Indeed, we would contend that for these more advanced transformations it is impossible to make them automatic, and that the role of the refactoring tool becomes one of a refactoring assistant, which can provide support for various aspects of the refactoring process, rather than a completely automated process. Perhaps this should be no surprise, as this is the case in machine proof, where theorem-provers and proof assistants co-exist, and there is more than a little in common between meaning-preserving refactoring and proof. We therefore expect that our work will take us towards more complex, user-driven, interactions.

We also see in the work that we report there is a substantial investment in infrastructure in any tool building of this sort. While it may not be evident from the high-level report of the Eclipse integration that we provided, the project report [26] shows this was not a trivial, or even a straightforward exercise, and considerable work remains to be done. Nevertheless we expect to contribute our refactoring tools to the general Erlide project, which shows great promise.

On the same theme we and the team from Eötvös Loránd University hope to evolve a common infrastructure between our two systems, so that user can take advantage of the two in a seamless way. This common infrastructure will also allow us to test the two systems against each other.

References

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

[2] R. Carlsson., Erlang Syntax Tools, `http://www.erlang.org/doc/apps/syntax_tools-1.5.5/`.

[3] D. Byrne. Integrating Java and Erlang. `http://www.theserverside.com/tt/articles/article.tss?l=IntegratingJavaandErlang`.

[4] E. Ciccarelli. An Introduction to the Emacs Editor. Cambridge, Massachusetts: MIT AI Lab., AIM-447, 1978.

[5] E. Clayberg and D. Rubel. Eclipse: Building Commercial-Quality Plug-ins. Addison Wesley, 2006.

[6] T. Corbat, L. Felber and M. Stocker. Refactoring Support for the Eclipse Ruby Development tools. Diploma thesis, Univ. of Applied Sciences, Rapperswil, Switzerland, 2006.

[7] Eclipse project. `http://www.eclipse.org/`.

[8] Erlide. `http://erlide.sourceforge.net/`.

[9] M. Fowler, *et. al. Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[10] L. Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. Eclipse Magazin, 2006.

[11] L. Gorrie. Distel: Distributed Emacs Lisp (for Erlang). In *The Proceedings of Eighth International Erlang/OTP User Conference*, Stockholm, Sweden, November 2002.

[12] S. Holzner. Eclipse. O'Reilly, 2004.

[13] Z. Horváth et al. Refactoring Erlang Programs. `http://plc.inf.elte.hu/erlang/`

[14] H. Li, C. Reinke, S. Thompson. Tool support for refactoring functional

programs. *Proceedings of the ACM SIGPLAN workshop on Haskell*, Uppsala, Sweden, 2003.

[15] H. Li, S. Thompson, C. Reinke. The Haskell Refactorer, HaRe, and its API. Electronic Notes in Theoretical Computer Science **141**(4) (2005) 29–34

[16] H. Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, UK, September 2006.

[17] H. Li, S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006

[18] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, T. Nagy. Refactoring Erlang programs. In: The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden (2006) http://www.erlang.se/euc/06.

[19] H. Li, S. Thompson. Testing Erlang Refactorings with QuickCheck. In Proceedings of IFL2007, Freiburg,, 2007.

[20] H. Li, S. Thompson. Tool Support For Refactoring Functional Programs. In: In Partial Evaluation and Program Manipulation (PEPM'08). San Francisco, California, USA (2008).

[21] H. Li and S. Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In P. Achten, *et. al.* eds, *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming*, The Netherlands, 2008.

[22] L. Lövei, Z. Horváth, T. Kozsik, R. Király. Introducing Records by Refactoring In: Proceedings of the 2007 SIGPLAN Erlang Workshop, Freiburg, Germany, Oct 2007

[23] L. Lövei, *et. al.* Refactoring Erlang programs. To appear in: Periodica Polytechnica – Electrical Engineering (2007)

[24] T. Mens and T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2) (2004)

[25] T. Nagy, A. Víg. Erlang refactor tool. Master thesis, Eötvös Loránd University, Budapest, Hungary, 2007.

[26] G. Orosz. The Eclipse integration of the Wrangler Erlang refactor tool. Report, Computing Lab, Univ. of Kent, 2008.

[27] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, *et. al.* eds, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[28] C. H. Roy and R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Candada, 2007.

[29] R. Szabó-Nacsa, P. Diviánszky, and Z. Horváth. Prototype environment for refactoring Clean programs. In *CSCS 2004, Szeged, Hungary*,, 2004.

[30] VIM Editor homepage. http://www.vim.org/. 2006.06.01.

[31] M. Weiser. Program Slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.