Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study

Dániel Horpácsi
Eötvös Loránd University
daniel-h@elte.hu

Judit Kőszegi
Eötvös Loránd University
koszegijudit@elte.hu

Zoltán Horváth
Eötvös Loránd University
hz@elte.hu

Widely used complex code refactoring tools lack a solid reasoning about the correctness of the transformations they implement, whilst interest in proven correct refactoring is ever increasing as only formal verification can provide true confidence in applying tool-automated refactoring to industrial-scale code. By using our strategic rewriting based refactoring specification language, we present the decomposition of a complex transformation into smaller steps that can be expressed as instances of refactoring schemes, then we demonstrate the semi-automatic formal verification of the components based on a theoretical understanding of the semantics of the programming language. The extensible and verifiable refactoring definitions can be executed in our interpreter built on top of a static analyser framework.

1 Introduction

Refactoring is a widely used technique that improves non-functional properties of source code without affecting dynamic semantics. Tool-assisted refactoring transformations must be trustworthy if programmers are to be confident in applying them on arbitrarily extensive and complex code in order to improve style or efficiency. Unfortunately, currently available, widely used refactoring frameworks do not guarantee the correctness of the transformations they implement, thus application of these might introduce undesired bugs to the system. In an ideal setting, such frameworks should provide formal guarantees, and they also should be extensible, so that programmers can compose their own program transformations according to their need. In this paper, we demonstrate our latest results in defining trustworthy refactoring definitions, with particular attention to a fairly complex case study which is intended to demonstrate the applicability of our approach to verifiable refactoring definition. We defined the refactoring formalism for the functional programming language Erlang, but we did so with other languages and paradigms in mind; the presentation is as independent of the functional object language as possible.

The main contributions of this paper are:

- Definition and application of language-agnostic refactoring schemes,
- Decomposition of a complex refactoring transformation into a series of prime refactorings,
- Definition of the prime refactorings in terms of instantiations of refactoring schemes,
- Demonstration of the verifiability of refactoring schemes and scheme instances based on the formal semantics of the object language.

The rest of the paper is structured as follows. Section 2 summarises the background of the work and at the same time it gives a brief overview of related results. Section 3 explains the core ideas behind our approach to refactoring definition and verification, Section 4 presents the definition of a complex case study in our formalism, and then Section 5 demonstrates the verifiability of the case study. Finally, Section 6 concludes.

2 Background and related work

In classic terminology, refactoring is a behaviour-preserving program transformation. From a more theoretical viewpoint, refactoring is a transformation on some model of the program, where applying the transformation to the model of a particular program should result in a semantically equivalent model with respect to some definition of formal semantics and semantic equivalence.

2.1 Program representation

There are several ways to represent the syntax and static semantics of program source code, but in most cases, an extension of the abstract syntax tree (AST) is used. This is beneficial for at least two reasons: 1) the syntax is easiest captured and manipulated by using the syntax tree, and 2) the formal semantics of programs is usually defined on the AST, thus any models that include (or can be turned into) the AST are amenable to semantic equivalence checking. Formal semantics usually captures the operational behaviour of the code, while semantic equivalence states that for any input, the original and resulting programs produce the same output.

The role of semantic program models is to capture the syntax and static semantics of the code in a compact yet efficient way. This is usually achieved via static semantic analysis [20] that enriches the abstract syntax tree with semantic nodes and edges representing context-sensitive properties. A good program representation simplifies code understanding, further static analysis as well as program transformation. The definition of a refactoring transformation highly depends on the abstraction level and richness of the used program model: the more details the model captures, the easier to express preconditions and transformation steps. In the ideal case, the model uses the same abstractions as the refactoring specification.

2.2 Transformation definition

Essentially, the interpretation of a refactoring transformation is a deterministic relation from programs to programs, or more precisely, from program models to program models. When the program model is the abstract syntax tree, this relation can practically be described with (strategic) term rewriting. In case the model happens to be a graph, graph rewriting can be applied instead. Indeed, the two classic approaches to formally defining refactoring are term rewriting and graph rewriting, and different refactoring specification formalisms adapt the core ideas of rewriting.

With the emerging use of tool-assisted refactoring, the interest in specification and verification of transformations has been constantly increasing and guarantees are needed for users to trust the complex changes carried out by tools. The abstractions for defining refactoring as well as the levels of correctness guarantees are varying, but almost every approach incorporates the fundamental work of Opdyke [14] that suggests refactorings be composed of basic steps called micro-refactorings. Simpler transformations are easier to read, write and to verify, but on the other hand, decomposition of extensive refactorings to simple steps requires experience and considerable effort.

Strategic term rewriting. Context-free conditional rewrite rules and functional strategies [2] are widely used to implement program transformations and stuctured data transformations in general. Furthermore, Bravenboer and Olmos show that by adding dynamically defined rewrite rules into the system [1], context-dependent transformations [13] are also definable.

Beyond any questions traversal programming is an expressive and exciting paradigm, but as Lämmel and others [11] point out in their comprehensive study, error-free use of strategy combinators requires expertise, not mentioning the difficulties of formal verification (the termination property of a complex strategy alone is a considerably difficult problem). The paper characterises typical mistakes in strategic programming, and one of their findings is that errors mostly stem from mixing up selection of terms of interest (their type and pattern), keeping track of the origin of data, checking side conditions and doing actual transformation.

Graph rewriting. Semantic program graphs capture the binding structure, the data and control flow relations in the program, while they may also depict properties of specific program symbols. It is apparent that semantics-aware, verifiable transformations can be specified with graph rewriting [12] as well, but the graphical descriptions of graph rewrite rules are relatively complex compared to concrete syntax patterns. In addition, matching a graph pattern to a semantic program graph is computationally more complex than matching a first-order term pattern to a term. Since the graphical format of rules is representation-dependent and rather complicated, this system is less likely to be used by users to define their own refactorings. Some systems use a graph model, but express the context-sensitive rewrite rules with a special textual representation, e.g. Padioleau et al. [15] use a transformation language incorporating semantic conditions into the textual patterns.

Refactoring languages. Designing domain specific languages for refactoring programming is a well-known idea, there are related results for different object languages with different representations. Some of these define the entire code transformation logic including term-level rewriting, while some only offer a formalism for composing atomic steps in a convenient way. Leitdo [9] gives an executable, rewrite-based refactoring language with expressive patterns, Verbaere [21] proposes a compact, representation-level formalism for executable definitions. These formalisms are expressive and language-independent, but at this generality they cannot support correctness checks for refactoring definitions. For Erlang, Li and Thompson [10] define an API for describing prime refactorings and a feature-rich language for interaction-aware composition, but formal verification is not addressed in their work either.

2.3 Verified refactoring

For object-oriented languages, Schaefer [18] introduced a system in which he reasoned about semi-formal definitions of a set of basic refactorings. The idea of using locks and language extensions instead of preconditions is exciting and promising, but the expressiveness is limited due to the lack of custom rewritings, and the proofs are mostly informal rather than formal. Roberts [16] applies a different definition style, with an emphasis on the side-conditions and proper composition of the base refactorings. However, neither of them provides formally verified or executable definitions. There are some results [19] in defining provably correct refactorings for simple languages, some mechanised proofs even for real-world refactorings [6] are available, but none of these allow for defining custom transformations and provide automatic verification for those.

We cannot talk about verified refactoring without clarifying object language semantics and the definition of semantic equivalence. Matching logic and reachability logic provide a formal system [7] for defining formal semantics for programming languages, doing model checking, and also for expressing semantic equivalence. Although they demonstrate how to use the framework for proving equivalence of programs (and program patterns), they do not employ the system for checking transformation definition correctness.

3 Our approach: decomposition to refactoring schemes

In our prior work [8], we introduced novel abstractions for executable and verifiable refactoring definitions. Besides incorporating the basic idea of decomposition to micro-refactoring, we use a semantic program graph representation to define transformations in terms of strategic term rewriting rules extended with language-level semantic conditions. Unlike strategic programs, our definitions separate analysis, target selection (object or term of interest), side condition checks and actual transformation. Our transformation language includes the well-known basic strategies, but in order for a transformation to be verifiable, it has to be expressed with the semantic schemes we provide for refactoring programming. These schemes depict special patterns in the program graph (selection), define some preconditions for the refactoring (condition), and control the rewrite rules that instantiate the scheme (transformation). Semantic schemes can be seen as verified algorithmic skeletons (or rewrite strategies), whose instances can be automatically verified for correctness.

3.1 Prime and composite refactoring functions

Refactorings are defined with functions, either prime (specifying a non-decomposable refactoring step) or composite (composing refactoring steps together). Refactoring functions have a target (the object of interest) determining a reference to a syntactic or semantic element in the program, and a return value characterising the result. Refactoring functions can take an arbitrary number of arguments of primitive types or node references, and they can define local variables by matching or with binding conditions.

Each prime refactoring function is an instance of a refactoring scheme, which provides a declarative description of the desired change: control is encoded into the interpretation of the scheme, while local variables in prime functions are all single-assignment. On the other hand, composite functions are imperative descriptions controlling application of other refactoring functions with target selectors (query functions for finding program elements of interest), sequential composition and iteration.

3.2 Schemes

Extensive code transformations can be expressed with traversal strategies, strategy combinators and complex semantic queries. Nevertheless, our goal is to make every transformation not only executable but also automatically verifiable. To achieve this, we hide semantics-based conditions and control behind pre-verified schemes. They can be understood as complex strategies in traversal programming, but in fact they are much more: schemes define the format of their parameter rewrite rules and may inspect the elements of the rules in order to define the compound strategy they carry out.

Since complex data and control dependencies are present among the various elements of the program, some transformations can only be correct (i.e. behaviour-preserving) if all the dependencies are handled properly when the origin of the dependency changes. Schemes make sure that the transformation will reach out to all code locations that might be affected, and also make sure that the changes made are consistent. Intentionally, schemes hide the complexity connected to semantics-based term selection and side conditions, while at the same time they fully control the application of rewrite rules by relying on these semantic connections. Schemes are instantiated with a series of conditional term rewrite rules, which are expressed in the concrete syntax of the object language, and they may refer to pre-defined semantic functions and predicates. These latter provide access to the program representation with an interface that resembles object language level concepts, allowing anyone knowing the object language to read and write refactoring definitions.

Local refactoring. The simplest scheme transforms a single sub-tree (or sub-term) in the program, and there is no control or conditions built into this strategy. Local refactoring simply applies the rewrite rule it takes directly to the program element selected for transformation.

Data-flow and control-flow driven refactoring. One of the core ideas of schemes is that dependencies connect program elements that shall be changed consistently. Data-flow induces data dependency, so when an element of a data-flow chain is changed, it entails the need for adjusting the rest the chain. We have two schemes that can be used for refactoring data-flow chains: *forward data-flow*, which starts from the data origin and visits references, and *backward data-flow*, which first modifies the data reference and then compensates data sources accordingly to keep consistency.

Binding driven refactoring. Names can induce data as well as control dependencies, and in most cases, when changing binding definitions, references have to be adjusted in order to preserve behaviour. Since our case study object language is Erlang, we identified refactoring schemes for *refactoring variables, functions, records and types*. Any semantic objects that can be given a name can be treated the same way, and obviously, in different programming languages, the set of these will differ.

Introduce binding. Introducing abstractions into the program is special in some sense, because although it involves changes at two different locations, one is merely addition and only the other is modification. Schemes of this kind introduce a name and, at the same time, they rewrite a piece of code to use the new binding. In fact, the change is the use of a name, and the compensation of this change is introducing the binding. Semantically, not only the binding is added to the code, but inherently the flow and dependency graphs are extended, too. Currently, we have schemes that *introduce variables and functions* by extracting expressions.

3.3 Correctness

We express refactoring correctness in terms of a set of equivalence formulas. In our previous work [8] we presented how to use a proof system to prove refactorings whose correctness can be expressed by the equivalence of two expression patterns under a given condition. We reduced the equivalence property of the two expression patterns to a correctness property of an aggregated program constructed by the two expression patterns (according to [4]), then we applied the language-independent, general-purpose proof system to automatically check the validity of our property.

According to our terminology, refactoring correctness is defined with respect to a formal semantics of the object language and an equivalence relation. We formalised a nearly complete, sequential and deterministic sub-language of Erlang with matching logic formulas, used throughout our proofs. Although the presentation of the entire language definition is beyond the scope of this paper, we include the semantic rules we apply when demonstrating the case study verification. We also define a suitable equivalence formula for our proofs in Section 5.

In order to verify refactorings, we turn refactoring functions into sets of conditional equivalence formulas. For local refactorings, this means simply treating the conditional rewrite rule as a pair of patterns; for strategy-combined rewritings, we face a more complex issue that has to glue rewriting, context and control. We split the verification problem in half: check that the scheme is correct under some assumptions (i.e. a contract), and then prove that the instantiation of the scheme satisfies those assumptions. Typically, contracts are equivalence formulas constructed from elements of the instantiation rules, while the verification of the scheme itself is a structural induction proof with base cases proven by the contract.

4 Decomposition and definition of the case study

The methodology of applying decomposition and schemes for defining refactoring is best demonstrated through a meaningful case study. We explain the decomposition process and the role of schemes as building blocks by formally specifying a well-known and fairly complex function refactoring: *generalise function definition*. As object language, we use Erlang [3], an impure, eagerly evaluated, dynamically typed functional programming language.

4.1 Informal specification

Our case study "generalise function" is a refactoring transformation that turns some value (i.e. a sub-expression within the function body) into a function parameter, thus making the function more abstract. The generalization increases the function arity by one, meaning it will take an extra formal argument compared to the original signature — this requires that this generalised signature is not defined in the code yet, which is one of the side-conditions of this transformation. In practice, there are two well-known realisations of this refactoring:

1. Generalise the function and then create a fall-back version with the original arity, where the fall-back function simply invokes the newly generalised version by passing as extra argument the extracted expression. This way, call sites to the original function can be left unchanged, since by calling the fall-back function their behaviour remains the same.
2. Change the call sites so that they pass the extracted expression as extra argument to the new, generalised function. This variant does not duplicate the function, but may affect a large number of code locations if there are a number of references to the generalised function.

The first variant is more local as the effect of the transformation remains in the module, while the second variant might reach out to other modules calling the generalised function. In both versions, the expression in question is moved from the function body to the call sites, thus the transformation has to make sure that the binding structure present in the expression is not affected by the relocation. Also common in both variants that they refactor variable and function objects in a general manner, which makes their definition pretty similar. In fact, the first one is a bit more challenging as it both changes the original function and adds a new one, which have to be kept semantically consistent, so we put our focus on defining the first variant of the refactoring.

Example. In order to demonstrate the behaviour of this transformation, we present a small piece of code and generalise the function f by lifting the constant 2 into a function argument. The presented example is intentionally overly simple, yet it shows how the abstractions are extended and changed, which sheds some light on what kind of schemes might be needed for ensuring consistent modification.

```
f(X) -> begin X * 2 end.          % function to be generalised
g(X) -> f(X+1).                   % a reference
```

Listing 1: Original code

The refactoring generalises the function by adding a new parameter to it and replacing the constant value with the new parameter in the body. At the same time, it creates a copy of the function that simply calls the generalised one with the original constant value. It might seem useless in this example, but because the expression we relocate may have side-effects, it should get encapsulated by a lambda function (denoted with the `fun` keyword in Erlang) and its application — this encapsulation enables the refactoring to keep the order and number of side-effects.

```
f(X, Y) -> begin X * Y() end.     % new, generalised function
f(X)    -> f(X, fun() -> 2 end). % invokes the new one
g(X)    -> f(X+1).               % callee unchanged
```

Listing 2: Refactored code

After carrying out function generalization, new names and signatures appear: a “new” f taking two arguments gets introduced, where the last argument is the newly introduced variable that takes the extra function parameter. In the next section, we elaborate on how the introduction and manipulation of these abstractions can be split into multiple stages.

4.2 Decomposition

By decomposition, we mean expressing a complex refactoring transformation in terms of smaller, simpler refactoring steps. This requires additional effort compared to specifying a transformation as a whole, but it pays off: smaller steps are easier to read, write, and are more easily checked for correctness. In order to decompose a refactoring, we need to understand how it affects language objects, data-flow and control-flow, clarifying how it is boiled down to simpler yet behaviour-preserving steps. Note that in many cases, there are multiple possible decompositions, which may differ in complexity and verifiability.

Avoiding detached refactoring. When designing decomposition, we avoid hidden or detached changes, i.e. those that introduce or modify dead code. These are easy to reason about since they are not part of the control-flow nor the data-flow (their modification is not observable from the semantic point of view), but relating detached changes to the original program requires overly complex syntactic or semantic conditions. In the most difficult case, side conditions involving dynamic semantic equivalence of arbitrary expressions might be needed, which we do not support in our formalism. As a matter of fact, we do not incorporate the formal semantics of the object language into the refactoring execution. When checking equivalence is inevitable, the condition might refer to a more restrictive condition that ensures syntactic equivalence.

The “generalise function” refactoring could be seen as two big, standalone steps: a (detached) refactoring that creates the generalised function definition, plus another one, which rewrites the original function as an application of the generalised one. Needless to say, this would pose a need for a complex precondition for the second step, namely a formula ensuring that calling the generalised function with the originally selected expression as extra argument is semantically equivalent to the original function body. Rather than composing the complex transformation of two independent transformations, we are going to specify it as a composition of several refactoring scheme instances.

Scenario. By building on the refactoring schemes identified in the previous section, we divide the case study refactoring into prime refactoring transformations that are easier to understand and verify. It is apparent that the complex refactoring will introduce new abstractions: a new function abstraction is created for the generalised instance, and a variable abstraction is created for the new parameter holding the value of the generalised expression. Rather than copying the function and then inlining, or adding an unused parameter and then integrating it into the body, we operate with slight yet completely behaviour-preserving changes to the abstractions. In each step, we highlight the term of interest we rewrite with a micro-refactoring (also, on the arrows, we identify the refactoring function and its arguments).

```
f(X) -> begin X*2 end.
g(X) -> f(X+1).
```

$\xrightarrow[\textit{(change expression)}]{\textit{wrap expression}}$

$\longrightarrow$
```
f(X) -> begin X*fun() -> 2 end() end.
g(X) -> f(X+1).
```

$\xrightarrow[\textit{(introduce function)}]{\textit{extract expression to function (z)}}$

$\longrightarrow$
```
z(X) -> begin X*fun() -> 2 end() end.
f(X) -> z(X).
g(X) -> f(X+1).
```

$\xrightarrow[\textit{(introduce variable)}]{\textit{extract expression to variable (Y)}}$

$\longrightarrow$
```
z(X) -> begin Y=fun() -> 2 end, X*Y() end.
f(X) -> z(X).
g(X) -> f(X+1).
```

$\xrightarrow[\textit{(change variable scope)}]{\textit{outer variable}}$

$\longrightarrow$
```
z(X) -> Y=fun() -> 2 end, begin X*Y() end.
f(X) -> z(X).
g(X) -> f(X+1).
```

$\xrightarrow[\textit{(change function)}]{\textit{variable to function parameter (Y)}}$

$\longrightarrow$
```
z(X, Y) -> begin X*Y() end.
f(X) -> z(X, fun() -> 2 end).
g(X) -> f(X+1).
```

$\xrightarrow[\textit{(change function signature)}]{\textit{rename function (z to f)}}$

$\longrightarrow$
```
f(X, Y) -> begin X*Y() end.
f(X) -> f(X, fun() -> 2 end).
g(X) -> f(X+1).
```

After performing 6 small refactoring steps, we arrive at the same result we had in our example presented in the previous section, which is the core idea behind micro-refactoring. In the following section, we are going to define each of these refactoring transformations in our specification formalism, and we also define a composite refactoring function that controls the application of these constituent steps.

4.3 Formal definition

In this section, we give a formal specification for the steps used in the decomposition. Our definition includes two composite function definitions and six prime refactoring functions derived from multiple schemes. We omit the formal definition of the selector "function_part", which queries the lambda function from the result of wrapping, and we also use some semantic functions that are not defined formally in this presentation.

For the syntax and semantics of the refactoring formalism, we refer to the comprehensive introduction presented in [8]. Roughly summarising, composite functions define a refactoring sequence, whilst prime functions instantiate schemes with rewrite rules applied to the definition and the references of some data or name. In any refactoring function, "THIS" is a local variable that holds a reference of the object of interest, that is, the program element the refactoring function is applied to (the refactoring target).

4.3.1 Composite refactoring functions

The main refactoring function is called `generalise_function` and it takes one argument determining the name of the new variable added to the function signature. It is merely a sequential composition of the rest of the refactoring functions, though it refers to two semantic functions as well: `function` associates the containing function with any syntactic element, while `name` simply returns the name of the function.

```
REFACTORING generalise_function(ParamName)
DO
    THIS.wrap()
    THIS = THIS.function_part()
    Old = function(THIS)
    Name = name(Old)
    Params = function_params(Old)
    New = Old.body().extract_to_function(tmp, Params)
    Var = THIS.extract_to_variable(ParamName)
    Var.to_function_parameter()
    New.rename_function(Name)
```

The result of one transformation can be the target or argument to other functions, like in pipelines. When a component step fails, the entire composition fails, and all intermediate results are rolled back. Although incomplete composite functions are correct as all composed steps are refactorings alone, intermediate changes may be undesired.

We introduce another composite refactoring function, `to_function_parameter`, which is targeting the variable matching created by a preceding step, and it lifts the new variable into a function parameter. This involves two different steps: it iterates lifting between scopes until the variable reaches the scope of the function (at this point the iteration construct will terminate successfully), and then it lifts the function-level variable to the parameter list.

```
REFACTORING to_function_parameter()
DO
    ITERATE THIS.outer_variable()
    function(THIS).var_to_param(THIS)
```

Note that by resembling method invocation in object-oriented languages, we allow refactoring functions be applied to language elements specified either with a variable or by using selector functions.

4.3.2 Prime refactoring functions

Prime refactorings implement micro-refactorings. They are expressed with refactoring schemes that control the application of the rewrite rules used for scheme instantiation. We define a local refactoring for making an arbitrary expression "movable", as a result of which the expression is wrapped into a lambda function. Though being a simple change, it has complex conditions: it requires that the expression does not bind any variables that are used outside the expression (predicate `non_bind`), while another condition binds a metavariable to hold the variable names that are free in the expression (referred to by the expression, but bound in the context).

```
LOCAL REFACTORING wrap()
    E
    ------------------------------
    (fun(Vars..) -> E end) (Vars..)
WHEN
    Vars.. = free_vars(E) AND non_bind(E)
```

We define two instances of the variable introduction scheme for introducing and lifting the new parameter of the generalised function. Both define the syntactic construct creating the binding, determine the place (the scope) of the binding, and they also specify the rewrite rule that will transform the target expression to use the newly introduced binding. Since new variable bindings can be placed either in the current scope or in an outer scope, this has to be decided in the instantiation of the scheme, while the conditions regarding name clash should be handled inherently. With this, we can express both introduction and lifting with the same scheme, and in the second one, the variable name is coming from the already present binding rather than from the refactoring argument. Syntactic noise (e.g. dead scope) introduced by intermediate steps can be removed by dedicated clean-up refactorings at the end of the process, however, we do not include clean-up transformations in this presentation.

```
INTRODUCE VARIABLE
  extract_to_variable(Name)
DEFINITION IN SCOPE
    Name = E
REFERENCE
    E
    ----
    Name
```

```
INTRODUCE VARIABLE
  outer_variable()
DEFINITION IN OUTER SCOPE
    Name = E
REFERENCE
    Name = E
    --------
    Name
```

We use the function introduction scheme for creating the fall-back function. Unlike in variable introduction, function definition placement is not an issue (the module name space is flat in Erlang), it does not matter where in a module a function is placed. The scheme implementation will append the new definition to the file.

```
INTRODUCE FUNCTION extract_to_function(Name, Params..)
DEFINITION
    Name(Params..) -> E .
REFERENCE
    E
    --------------
    Name(Params..)
WHEN is_subset(free_vars(E), vars(Params..))
```

Perhaps the most interesting components in the compound refactoring are the ones transforming the function and its signature. The function refactoring scheme transforms the function as well as its references by applying the supplied rewrite rules on the definition and on all kinds of references, including calls, name references and directives.

```
FUNCTION REFACTORING var_to_param(X)
DEFINITION
    (Args..) -> X = E, Body..
    -------------------------
    (Args.., X) -> Body..
REFERENCE
    (Args2..)
    ------------
    (Args2.., E)
WHEN pure(E) AND closed(E)
```

A special case of function refactoring is function signature refactoring, which only transforms the head of the function definition and its refereces. As we demonstrated in our previous paper [8], this scheme can be used as well for renaming a function and to restructure or reorder its arguments.

```
FUNCTION SIGNATURE REFACTORING rename_function(NewName)
    Name(Args..)
    ---------------
    NewName(Args..)
```

5 Verification of the case study

In this section we demonstrate the formal verification of some components of our case study defined in Section 4.3. In order to do so, we incorporate our definition of formal semantics and formula of semantic equivalence designed for our example object language, Erlang.

5.1 Equivalence of Erlang expression patterns

As mentioned already, correctness of a refactoring definition is reduced to an equivalence problem of expression patterns, so at first, we need to find a sensible and appropriate definition of semantic equivalence between Erlang expression patterns. Since semantic equivalence can be defined in different granularities (determined by the details we take into account about the program elements and their execution environment), the way we formalise equivalence has a great influence on what patterns will be regarded as being equal. For instance, equivalence can be defined to only compare the values of expressions, but it might also expect the expressions to have the same series of side effects or to show the same behaviour on exceptional inputs. Although the configuration in our object language semantics is rather fine-grained (i.e. it captures module structure with module attributes and function definitions, variable environment and side effects), in our proofs we abstract on this and make equivalence requirements only on the code part and its variable environment. Also, for some proofs, this has to be relaxed to include the side effects:

$$\langle \langle\rangle_{\mathsf{code}}\ \langle\rangle_{\mathsf{env}}\ \langle\rangle_{\mathsf{side\_eff}} \rangle_{\mathsf{cfg}}$$

Equivalence has to imply the following property: the expressions are interchangeable in any program context so that the meaning of the entire program is preserved. The equivalence specification of Erlang expression patterns should ensure that any concrete expression pairs instantiated from the patterns are evaluated to the same value, their evaluation has the same impact on the variable environment, and the same side effects occur during their evaluation.

This equivalence relation can be specified with matching logic formulas. Matching logic allows us to specify patterns over program configurations, and additional constraints can be added to the configuration expressed with regular first-order logic expressions. For equivalence checking, the configuration of the equivalence problem is composed of two configurations of the object programming language. We specify equivalence of Erlang program patterns with two formulas: one regarding successful evaluation and one for failure.

The first formula says that the two patterns can be rewritten to the same form (are 'joinable'), side effects occurred in the same number and order, and variable mappings are identical, too. The second formula matches pairs of program states where both executions terminate with an exception (the types of exceptions do not have to be the same, though). Unlike Erlang, the sub-language we formalized does not provide exception handling, in our model exceptions will simply terminate the execution and ignore the rest of the program.

$$\mathtt{Eq} \equiv \left\{ \left\langle \begin{matrix} \langle\langle \mathit{Code}\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} & \langle \mathit{SE}\rangle_{\mathsf{side\_eff}}\rangle_{\mathsf{cfg}} \\ \langle\langle \mathit{Code}\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} & \langle \mathit{SE}\rangle_{\mathsf{side\_eff}}\rangle_{\mathsf{cfg}} \end{matrix} \right\rangle_{\mathsf{eq}}, \left\langle \begin{matrix} \langle\langle\rangle_{\mathsf{code}} & \langle\varepsilon_1\rangle_{\mathsf{env}} & \langle \mathit{SE}, \mathtt{\#exception}\rangle_{\mathsf{side\_eff}}\rangle_{\mathsf{cfg}} \\ \langle\langle\rangle_{\mathsf{code}} & \langle\varepsilon_2\rangle_{\mathsf{env}} & \langle \mathit{SE}, \mathtt{\#exception}\rangle_{\mathsf{side\_eff}}\rangle_{\mathsf{cfg}} \end{matrix} \right\rangle_{\mathsf{eq}} \right\}$$

5.2 Proof system

The general-purpose reachability proof system, referred to in our previous publication [8], can generate proofs automatically, but it can only prove partial equivalence and not full equivalence. Moreover, this generic proof system generates unnecessarily many branches, because it cannot take into account the specialities of equivalence proofs of deterministic programs. Since then, a new matching logic based language independent 5-rule proof system was published for checking full program equivalence of deterministic programs [5]. The inference rules are directly designed for equivalence proofs, making the reasoning higher-level, effective and readable. Moreover, it can also prove equivalence of non-terminating programs. Unfortunately, there is no tool support for this new Equivalence Proof System yet, but most formulas can still be automatically proved with the implementation of the general-purpose proof system.

In this paper, we apply the inference rules of the new, equivalence-focussed proof system, as it provides a more appropriate toolkit for phrasing our correctness checks. We are going to prove formulas of form $\vdash \varphi \Downarrow^{\infty} \mathtt{Eq}$, which express the derivability of an equivalence formula φ in the proof system with respect to a fixed formula set $\mathtt{Eq}$ characterizing the pairs of programs that are known to be equivalent. The formula is validated by applying inference rules of the proof system in the traditional bottom-up manner until every leaf of the proof tree is an instance of the AXIOM rule. For reasoning about our use case, we use the following three rules out of the five:

$$\textsc{Axiom}\ \frac{\varphi \in \mathtt{Eq}}{\vdash \varphi \Downarrow^{\infty} \mathtt{Eq}} \qquad \textsc{Conseq}\ \frac{\models \varphi \to \exists \tilde{x}.\varphi' \quad \vdash \varphi' \Downarrow^{\infty} \mathtt{Eq}}{\vdash \varphi \Downarrow^{\infty} \mathtt{Eq}}$$

$$\textsc{Step}\ \frac{\models \varphi_1 \Rightarrow_1^{*} \varphi_1' \quad \models \varphi_2 \Rightarrow_2^{*} \varphi_2' \quad \vdash \langle \varphi_1', \varphi_2' \rangle \Downarrow^{\infty} \mathtt{Eq}}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^{\infty} \mathtt{Eq}}$$

AXIOM states that every formula of the set $\mathtt{Eq}$ is derivable. CONSEQ allows to perform domain reasoning and prove a more general formula instead of the specific one. Last but not least, STEP allows to take arbitrary number of finite steps in each of the two program configurations. This rule weaves operational semantic rules into the reachability reasoning. For further details, see [5].

5.3 Proof sketch for the local refactoring

The correctness property for a local refactoring can be expressed by the equivalence problem of the two expression patterns given in the rewrite rule, extended with the conditions. This formula can be mechanically constructed, by inspecting the conditional rewrite rule defining the refactoring:

```
<pattern1>
-----------  WHEN <condition>
<pattern2>
```

The corresponding formula captures the equivalence of the patterns according to our previous definition of expression pattern equivalence. The condition is simply attached to the configuration pattern:

$$\varphi \equiv \langle\langle\langle \texttt{<pattern1>} \rangle_{\mathsf{code}} \langle \varepsilon \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}\ \langle\langle \texttt{<pattern2>} \rangle_{\mathsf{code}} \langle \varepsilon \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \rangle_{\mathsf{eq}} \wedge \texttt{<condition>}$$

Now let us demonstrate how the *wrap* local refactoring is verified in the proof system. First, let Ψ be the condition of the refactoring rule: $\Psi \equiv \mathit{Vars} = \texttt{free\_vars}(E) \wedge \texttt{non\_bind}(E)$. Note that italic upper-case names (like E) denote mathematical variables rather than program variables. So-called list metavariables (postfixed by "`..`", see [8]) become regular variables of the formula (e.g. *Vars*), because in matching logic we can capture them with types of sequences of elements (e.g. *VarList*). In the formula, a variable followed by a colon and a type name is a type-restricted variable; by default, all variables have the most general type allowed by the syntactic context. For explanation on the "followed by" ($\curvearrowright$) and the "hole" ($\square$) symbols we refer to the comprehensive overview of the $\mathbb{K}$ framework [17].

The proof (1) is read bottom up starting with line 3 (the proof goal). We perform STEP by using semantic rules and semantic lemmas on the second configuration of the goal, which leads to state 2. Then, by generalising the formula, we get the axiom: one of the equivalence specification formulas. Note that the cell of side effects is hidden as it remains empty during the proof.

$$\begin{array}{llllll}
1. & \vdash \left\langle \begin{matrix} \langle\langle \mathit{Code}\rangle_{\mathsf{code}}\ \ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\ \langle\langle \mathit{Code}\rangle_{\mathsf{code}}\ \ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \end{matrix} \right\rangle_{\mathsf{eq}} & \Downarrow^{\infty} & \mathsf{Eq} & \text{AXIOM} & \\
2. & \vdash \left\langle \begin{matrix} \langle\langle E\rangle_{\mathsf{code}}\ \ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\ \langle\langle E\rangle_{\mathsf{code}}\ \ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \end{matrix} \right\rangle_{\mathsf{eq}} \wedge \Psi & \Downarrow^{\infty} & \mathsf{Eq} & \text{CONSEQ}(1) & (1) \\
3. & \vdash \left\langle \begin{matrix} \langle\langle E\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\ \langle\langle \mathtt{fun}\,(\mathit{Vars})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(\mathit{Vars})\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \end{matrix} \right\rangle_{\mathsf{eq}} \wedge \Psi & \Downarrow^{\infty} & \mathsf{Eq} & \text{STEP}(2) &
\end{array}$$

To validate the application of the STEP rule, we need to prove the following reachability:

$$\langle\langle \mathtt{fun}\,(\mathit{Vars})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(\mathit{Vars})\rangle_{\mathsf{code}}\langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \ \Rightarrow^{*}\ \langle\langle E\rangle_{\mathsf{code}}\langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi$$

Starting with the left-hand-side configuration of the reachability, we rewrite the current state by applying either a proper semantic rule or a lemma until we get the right-hand-side configuration:

$$\begin{array}{lll}
\mathtt{cfg}_0 & : & \langle\langle \mathtt{fun}\,(\mathit{Vars})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(\mathit{Vars})\rangle_{\mathsf{code}}\langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{rule}_1 & : & \langle\langle F : \mathit{Fun}\,(\mathit{Exp}) \cdots\rangle_{\mathsf{code}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg\,\mathtt{isValue}\,(\mathit{Exp}) \Rightarrow \langle\langle \mathit{Exp} \curvearrowright F(\square) \cdots\rangle_{\mathsf{code}} \cdots\rangle_{\mathsf{cfg}} \\
\mathtt{cfg}_1 & : & \langle\langle \mathit{Vars} \curvearrowright \mathtt{fun}\,(\mathit{Vars})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(\square)\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{rule}_2 & : & \langle\langle \mathit{Vs} : \mathit{VarList} \cdots\rangle_{\mathsf{code}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathtt{lookup}\,(\mathit{Vs}) \cdots\rangle_{\mathsf{code}} \cdots\rangle_{\mathsf{cfg}} \\
\mathtt{cfg}_2 & : & \langle\langle \mathtt{lookup}\,(\mathit{Vars},\varepsilon) \curvearrowright \mathtt{fun}\,(\mathit{Vars})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(\square)\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{rule}_3 & : & \langle\langle V : \mathit{Value} \curvearrowright F : \mathit{Fun}\,(\square) \cdots\rangle_{\mathsf{code}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle F : \mathit{Fun}\,(V) \cdots\rangle_{\mathsf{code}} \cdots\rangle_{\mathsf{cfg}} \\
\mathtt{cfg}_3 & : & \langle\langle \mathtt{fun}\,(\mathit{Vars})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(\mathtt{lookup}\,(\mathit{Vars},\varepsilon))\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{rule}_4 & : & \langle\langle \mathtt{fun}\,(\mathit{Ps} : \mathit{PatternList})\,\mathtt{\text{->}}\,E\ \mathtt{end}\,(V : \mathit{Value}) \cdots\rangle_{\mathsf{code}}\langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\
 & & \Rightarrow \langle\langle \mathtt{matches}\,(V, \mathit{Ps}\,\mathtt{\text{->}}\,E) \curvearrowright \mathtt{restoreEnv}\,(\varepsilon) \cdots\rangle_{\mathsf{code}}\langle \mathtt{remove}\,(\mathtt{vars}\,(\mathit{Ps}),\varepsilon)\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\
\mathtt{cfg}_4 & : & \langle\langle \mathtt{matches}\,(\mathtt{lookup}\,(\mathit{Vars},\varepsilon), \mathit{Vars}\,\mathtt{\text{->}}\,E) \curvearrowright \mathtt{restoreEnv}\,(\varepsilon)\rangle_{\mathsf{code}} \\
 & & \ \langle \mathtt{remove}\,(\mathit{Vars},\varepsilon)\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{rule}_5 & : & \langle\langle \mathtt{matches}\,(V, \mathit{Ps}\,\mathtt{\text{->}}\,E) \cdots\rangle_{\mathsf{code}}\langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \mathtt{isMatching}\,(V, \mathit{Ps}, \varepsilon) \\
 & & \Rightarrow \langle\langle E \cdots\rangle_{\mathsf{code}}\langle \varepsilon \mathbin{++} \mathtt{getMatching}\,(V, \mathit{Ps}, \varepsilon)\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\
\mathtt{cfg}_5 & : & \langle\langle E \curvearrowright \mathtt{restoreEnv}\,(\varepsilon)\rangle_{\mathsf{code}} \\
 & & \ \langle \mathtt{remove}\,(\mathit{Vars},\varepsilon) \mathbin{++} \mathtt{getMatching}\,(\mathtt{lookup}\,(\mathit{Vars},\varepsilon), \mathit{Vars}, \mathtt{remove}\,(\mathit{Vars},\varepsilon))\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{lemma}_1 & : & \mathtt{remove}\,(\mathit{Vars},\varepsilon) \mathbin{++} \mathtt{getMatching}\,(\mathtt{lookup}\,(\mathit{Vars},\varepsilon), \mathit{Vars}, \mathtt{remove}\,(\mathit{Vars},\varepsilon)) \Rightarrow \varepsilon \\
\mathtt{cfg}_6 & : & \langle\langle E \curvearrowright \mathtt{restoreEnv}\,(\varepsilon)\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi \\
\text{lemma}_2 & : & \langle\langle E \curvearrowright \mathtt{restoreEnv}\,(\varepsilon)\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \mathtt{non_bind}\,(E) \Rightarrow \langle\langle E\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\
\mathtt{cfg}_7 & : & \langle\langle E\rangle_{\mathsf{code}}\ \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \wedge \Psi
\end{array}$$

Let us give a brief explanation for the semantic proof.

- Initially, we have a call for an anonymous function. The actual parameters of the call have to be evaluated to a value before the call itself is evaluated, so with $\texttt{rule}_1$ we force the evaluation of the parameters by moving them to the top of the cell ("heating" rule).
- The *Vars* $=$ $\texttt{free\_vars}(E)$ statement of the condition Ψ implies that the type of the *Vars* is list of variables. Variables are evaluated by looking up their associated values from the variable environment, by using the function `lookup` ($\texttt{rule}_2$). Note that `lookup` is not evaluated (the variables as well as the environment are symbolic), but we know that its return value has the type *Value*, so we can put it back to the argument of the call applying $\texttt{rule}_3$ ("cooling" rule).
- Then, $\texttt{rule}_4$ rewrites the function call to the `matches` intermediate structure used for program constructs where values have to be matched against patterns. The formal parameters of the anonymous function may shadow variables defined in the outer scope; therefore, all variables occurring in the patterns of the parameter list are removed from the variable environment. Besides, as the anonymous function opens a new scope, variable bindings inside the function should not have any effects on the containing scope. To reflect this, an extra operation is inserted into the `code` cell: `restoreEnv` is responsible for restoring the original variable environment after the call has been fully evaluated.
- We can apply $\texttt{rule}_5$ in the case when the value matches the pattern of the clause of the `matches` structure. In our case, the `isMatching` function of the rule constraint is evaluated to `true`, as *Vars* contains only variables that do not occur in the current variable environment (because of the `remove` operation), so they will match any value. Moreover, the length of the value list and the variable list are the same, because the `lookup` function does not change the length of the list given as parameter.
- Using $\texttt{lemma}_1$ we simplify the variable environment. The lemma can be derived from the definitions of contained functions. Informally, it states that if we remove variable assignments from the environment, and then we put back the same assignments, we obtain the original environment. Finally, $\texttt{lemma}_2$ expresses the fact that if we need to restore a variable environment after evaluating an arbitrary expression that does not introduce variable bindings, we can omit the `resotreEnv` operation containing the current environment in its parameter.

5.4 Proof sketch for a scheme instance

As clarified already, verification of a refactoring expressed as a scheme instance is two-phased: 1) the scheme has to be verified w.r.t. its contract on its instantiation, and 2) the instantiation has to be checked for conformance with the contract. In this section, we demonstrate the verifiability of the "extract to variable" refactoring function.

The variable introduction scheme

This scheme is characterized by the binding it creates (the name it introduces with the expression the name is bound to), and the rewrite rule that transforms the originally selected expression. When defining an instance of this scheme, the refactoring programmer, according to a pre-defined format, precisely specifies these elements. In this specific case, the introduction place is fixed to be the scope of the selected expression.

The generic format of the scheme is the following:

```
DEFINITION IN SCOPE
    <name> = <pattern1>
REFERENCE
    <pattern2>
    ----------
    <pattern3>
```

The contract for this scheme requires that

$$\texttt{begin <name> = <pattern1>, <pattern3> end} \equiv \texttt{<pattern2>}$$

under the side-conditions specified in the unfolding of the scheme.

It is apparent that the scheme can only be used for introducing variable bindings, but it also allows for modification of the reference site. According to the operational definition of the scheme, it is transformed into an extensive strategy relying on the semantic relation of scopes:

```
ON scope(THIS)
  E..
  -----------------------
  <name> = <pattern1>, E..
WHEN fresh(<name>) AND pure(<pattern1>) AND closed(<pattern1>)
THEN ON THIS
  <pattern2>
  ----------
  <pattern3>
```

This extensive scheme will transform the selected expression ("ON THIS") as well as its scope in order to carry out a consistent change. The expression is certainly a sub-expression of its scope, but the structural correspondence between them is defined by the semantic function "scope" — the transformation as a whole can be seen as a rewriting that involves a special semantic pattern. In order to prove the correctness of this extensive rewriting, we need to check that in any environment, with any particular nesting of the expression, the transformation is going to result in semantically equivalent code.

In order to cover every possible case, we rephrase the semantic relation as an inductively defined set by building on syntactic relations. For the sake of simplicity, let us now define *scope(X)* as the containing block of the expression, formally, *block(top_expression(X))*, where the relation *top_expression* is inductively defined with the direct sub-expression relation as its base case. Then we do the proof by induction on the structure of the relation *scope*, which in our case is a structural induction on the syntactic relationship between the expression and its containing block. Roughly speaking, we need to simulate all the possible embeddings of the expression in the block it is located inside, and compare these with their variants transformed by the instantiation rules.

The base case for this structural induction proof is characterised by the "empty environment", meaning the expression is a singleton top-level element of the block. Fortunately, this case is completely justified by the contract of the scheme. Then, by considering every single case where the expression is a sub-expression of another (characterising the inductive step of top_expression), we prove that regardless of the structure in the block, the equivalence relation holds. This step can be easily carried out based on the operational semantic rules that cover the entire language.

The scheme instance

In order to verify the scheme instance, we compose the equivalence formula by inspecting the instantiation rules. As for the "extract to variable" function, the following formula will express the validity of the scheme instance:

$$\vdash \left\langle \begin{matrix} \langle\langle E\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{begin}\, \mathit{Name} = E, \mathit{Name}\, \texttt{end}\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \end{matrix} \right\rangle_{\mathsf{eq}} \wedge\ \mathsf{fresh}(\mathit{Name}) \Downarrow^{\infty} \quad \texttt{Eq}$$

The formula can be proved with the same inference rules as the example proof we showed for the *wrap* refactoring.

6 Conclusion

We have presented a verifiable yet executable definition of a fairly complex refactoring transformation, aiming at demonstrating the applicability of our approach to refactoring formalisation. We have decomposed "generalise function", a well-known refactoring, in an unforeseen way into a couple of small semantics-preserving steps, each of which are semi-automatically verifiable. Our case study has justified that by decomposing composite transformations into instances of several simple refactoring scheme instances, refactorings of various complexities may be expressed and defined in a verifiable manner.

The core ideas of decomposition and refactoring schemes have been demonstrated for the Erlang programming language, but at the same time, major results of our work have been shown to be mostly language-agnostic and applicable to abstractions of other programming paradigms. In the future, we intend to compose a large number of refactoring definitions for a wide range of use cases and document their formal proof in full detail. Besides, we would like to define the refactoring language formally, both the operational behaviour and the method for turning refactoring definitions into automatically verifiable formulas.

7 Acknowledgement

The authors would like to express their sincere gratitude to Prof. Simon Thompson for his help and support in guiding the first steps in this research. We thank the reviewers for their constructive and helpful comments.

This research was supported through the New National Excellence Program of the Ministry of Human Capacities of Hungary.

References

[1] Martin Bravenboer, Arthur van Dam, Karina Olmos & Eelco Visser (2005): *Program Transformation with Scoped Dynamic Rewrite Rules*. Fundam. Inf. 69(1-2), pp. 123–178.

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas & Eelco Visser (2008): *Stratego/XT 0.17. A language and toolset for program transformation*. Science of Computer Programming 72(1–2), pp. 52 – 70, doi:10.1016/j.scico.2007.11.003.

[3] Francesco Cesarini & Simon Thompson (2009): *Erlang Programming*. O'Reilly Media, Inc.

[4] Stefan Ciobâca (2014): *Reducing Partial Equivalence to Partial Correctness*. In: *Proceedings of SYNASC '14*, IEEE, pp. 164–171, doi:10.1109/SYNASC.2014.30.

[5] Stefan Ciobâca, Dorel Lucanu, Vlad Rusu & Grigore Roşu (2016): *A Language-Independent Proof System for Full Program Equivalence*. *Formal Aspects of Computing* 28(3), pp. 469–497, doi:10.1007/s00165-016-0361-7.

[6] Julien Cohen (2016): *Renaming Global Variables in C Mechanically Proved Correct*. In Geoff Hamilton, Alexei Lisitsa & Andrei P. Nemytykh, editors: Proceedings of the Fourth International Workshop on *Verification and Program Transformation*, Eindhoven, The Netherlands, 2nd April 2016, *Electronic Proceedings in Theoretical Computer Science* 216, Open Publishing Association, pp. 50–64, doi:10.4204/EPTCS.216.3.

[7] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li & Grigore Roşu (2016): *Semantics-Based Program Verifiers for All Languages*. In: *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, ACM, pp. 74–91, doi:10.1145/2983990.2984027.

[8] Dániel Horpácsi, Judit Kőszegi & Simon Thompson (2016): *Towards Trustworthy Refactoring in Erlang*. In Geoff Hamilton, Alexei Lisitsa & Andrei P. Nemytykh, editors: Proceedings of the Fourth International Workshop on *Verification and Program Transformation*, Eindhoven, The Netherlands, 2nd April 2016, *Electronic Proceedings in Theoretical Computer Science* 216, Open Publishing Association, pp. 83–103, doi:10.4204/EPTCS.216.5.

[9] António Menezes Leitdo (2002): *A formal pattern language for refactoring of Lisp programs*. In: *Proceedings of CSMR '02*, IEEE Computer Society, Washington, DC, USA, pp. 186–192, doi:10.1109/CSMR.2002.995803.

[10] Huiqing Li & Simon Thompson (2012): *A Domain-specific Language for Scripting Refactorings in Erlang*. In: *Proceedings of FASE'12*, Springer-Verlag, Berlin, Heidelberg, pp. 501–515, doi:10.1007/978-3-642-28872-2_34.

[11] Ralf Lämmel, Simon Thompson & Markus Kaiser (2013): *Programming errors in traversal programs over structured data*. *Science of Computer Programming* 78(10), pp. 1770 – 1808, doi:10.1016/j.scico.2011.11.006.

[12] Tom Mens, Niels Van Eetvelde, Serge Demeyer & Dirk Janssens (2005): *Formalizing refactorings with graph transformations*. *Journal of Software Maintenance and Evolution* 17(4), pp. 247–276, doi:10.1002/smr.316.

[13] Karina Olmos & Eelco Visser (2005): *Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules*. In Rastislav Bodik, editor: *Compiler Construction*, *LNCS* 3443, Springer Berlin Heidelberg, pp. 204–220, doi:10.1007/978-3-540-31985-6_14.

[14] William F. Opdyke (1992): *Refactoring Object-oriented Frameworks*. Ph.D. thesis, University of Illinois.

[15] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall & Gilles Muller (2006): *Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers*. In: *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems*, PLOS '06, ACM, New York, NY, USA, p. 10, doi:10.1145/1215995.1216005.

[16] Donald Bradley Roberts (1999): *Practical Analysis for Refactoring*. Ph.D. thesis, University of Illinois.

[17] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[18] Max Schaefer & Oege de Moor (2010): *Specifying and Implementing Refactorings*. *SIGPLAN Not.* 45(10), pp. 286–301, doi:10.1145/1932682.1869485.

[19] Nik Sultana & Simon Thompson (2008): *Mechanical Verification of Refactorings*. In: *Proceedings of PEPM '08*, ACM, New York, NY, USA, pp. 51–60, doi:10.1145/1328408.1328417.

[20] Melinda Tóth, István Bozó & Zoltán Horváth (2011): *Reverse Engineering of Complex Software Systems via Static Analysis*. Lecture at 4th Central European Functional Programming School, Budapest, Hungary.

[21] Mathieu Verbaere, Ran Ettinger & Oege de Moor (2006): *JunGL: A Scripting Language for Refactoring*. In: *Proceedings of ICSE '06*, ACM, New York, NY, USA, pp. 172–181, doi:10.1145/1134285.1134311.