Acta Cybernetica — *online–first paper version* — pages 1–27.

Towards a Generic Framework for Trustworthy Program Refactoring*

Dániel Horpácsi[ab], Judit Kőszegi[b], and Dávid J. Németh[b]

Abstract

Refactoring has to preserve the dynamics of the transformed program with respect to a particular definition of semantics and behavioural equivalence. In general, it is rather challenging to relate executable refactoring implementations with the formal semantics of the transformed language. However, in order to make refactoring tools trustworthy, we may need to provide formal guarantees on correctness. In this paper, we propose high-level abstractions for refactoring definition and we outline a generic framework which is capable of verifying and executing refactoring specifications. By separating the various concerns in the transformation process, our approach enables modular and language-parametric implementation.

Keywords: refactoring, domain-specific language, refactoring methodology, formal verification

1 Introduction

The idea of refactoring is as old as high-level programming. A program refactoring [7] is typically meant to improve non-functional properties, such as the internal structure or the appearance, of a program without changing its observable behaviour. Tool support is necessary for refactoring in large-scale: it has to be ensured that program changes are complete and sound, the behaviour is intact and no bugs are introduced or eliminated by the transformations. Refactoring tools may carry out extensive modifications in large projects, which are hard to be performed or comprehended by humans.

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

[a]Project no. ED_18-1-2019-0030 (Application domain specific highly reliable IT solutions subprogramme) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

[b]ELTE Eötvös Loránd University, Budapest, Hungary. Faculty of Informatics, 3in Research Group, Martonvásár, Hungary. E-mail: daniel-h@elte.hu, koszegijudit@elte.hu, ndj@inf.elte.hu. ORCID: 0000-0003-0261-0091, 0000-0003-1915-4176, 0000-0002-1503-812X.

DOI: 10.14232/actacyb.284349

Generally speaking, the primary goal of a refactoring framework is to provide automatic tool support for behaviour-preserving (semantics-preserving) program rewriting. Additional general design goals may include support for interactive execution, multiple target languages or extensibility via user-defined transformations. Trustworthiness of refactoring implementations is usually backed by excessive amounts of testing, but research keeps looking for possibilities of formally specifying and verifying the correctness of program transformations. Whether a refactoring tool gets widely adopted highly depends on the extensibility and the trustworthiness of the solution.

Refactoring, from the programmer point of view, is an editor or dedicated tool function that helps modify the program in a well-understood way, increasing code quality. From the tool designer point of view, it is a complex process of creating and analysing a program model, locating elements of interest, and rewriting of the model to an equivalent model. In the typical case, these aspects of the process are mixed up, resulting in a hardly extensible, language-specific tool, which is unreasonably difficult to formally verify.

Our aim with this paper is to outline a generic design for a refactoring framework that has all the above-mentioned features: it uses executable and extensible definitions, supports multiple languages and enables semi-automatic formal verification. In particular, we present abstractions for program representation and refactoring definition, and we describe a language-parametric architecture that can be tailored to programming languages of different paradigms by supplying the formal definition of the language along with some refactoring schemes. The main contributions of this paper are:

- Design of a generic refactoring approach that supports executable and semi-automatically verifiable transformations via language-specific semantic predicates and refactoring schemes;
- Description of a language-parametric framework with language-specific artefacts and language-independent components, with a guideline on how the framework is tailored for a particular language;
- Testimonials of applying the above-mentioned framework to languages of two different programming paradigms.

We structure the rest of the paper as follows. First, we survey related work in Section 2, focusing on language-agnostic approaches and proven-correct refactoring. Then, we present high-level abstractions for defining transformations, which enable language-parametric implementation and formal verification (Section 3); this section is partly based on our previous work [11]. It is followed by the demonstration of the generic refactoring framework and its components in Section 4, and then we outline the main concerns with instantiating the framework for functional and object-oriented languages (Section 5). Finally, we conclude the paper after a brief discussion of our results.

2 Related work

There are various approaches to specifying and implementing refactoring, which vary in terms of the model they use for representing programs, in the abstraction level they use for specifying program semantics and program transformations, as well as in the level of guarantees they can provide on correctness. In this section, we overview the most important and influencing related work.

Preliminaries. In this paper, we use the terms *object language* or *target language* when referring to the programming language we aim to refactor. By *static semantics*, we mean the context-dependent part of the syntax (e.g. name binding and type information), as well as additional (possibly dynamic) semantic properties that can be statically extracted from, or approximated based on, the source code (such as purity, control flow and data flow). *Dynamic semantics* is the formal definition of program run-time behaviour, presumably given in small-step operational style or in reachability logic [24]. By *refactoring correctness* we mean that the transformation turns any program into a semantically equivalent program. This correctness property could be checked after each application of the refactoring (i.e. whether a particular execution of a transformation was correct), but in our research we aim at verifying the definition itself, i.e. formally reasoning about the transformation being correct applied to any program.

Compositional definition of refactoring. Although the abstractions for defining refactoring are varying from approach to approach, almost all solutions incorporate the fundamental work of Opdyke [21] that suggests refactorings be composed of basic steps called micro-refactorings. Simpler transformations are easier to read, write and to verify; on the other hand, decomposition of extensive refactorings to simple steps may require experience and considerable effort. Having said that, the compositional approach enables modularity both in definition, execution and in verification, and is therefore inevitable in designs where generality and formal verification are among the design goals.

Refactoring framework. Roberts [22] designed one of the first dedicated frameworks for implementing refactoring transformations. He states that refactoring tools should a) be completely automated; b) be provably correct and c) offer complex refactorings composed from primitive ones. We strongly share these fundamental design goals in our own approach, and in addition, we believe that even the primitive refactorings should be user-definable.

Widely used general-purpose programming languages have all gained their own language processor environments which support analysis and transformations on a model of the program, even functional programming languages, such as Haskell or Erlang. In fact, our approach born as a generalization of a refactoring framework [2] and its API designed for the Erlang programming language. The Erlang-specific solution was summarized along with a case study in [11].

Language-agnostic approaches. Language-independent specification of refactorings is an idea that pops up regularly, addressing the problem of semantics-preserving program transformation with generic program representations, analysis and traversal functionality. Lämmel [14] proposes a generic refactoring system based on Strafunski-style generic functional programming. It states that a refactoring can be described by a number of steps of the following kind: a) identification of fragments of a certain type and location; b) destruction, analysis, and construction; c) checking for pre- and postconditions and d) placing, removing or replacing a focus. Just like with work by Roberts, we strongly agree with Lämmel's thoughts about separation of concerns. In particular, we suggest that refactoring should be phrased as a composition of analysis and transformation, where transformation consists of pre-condition checks and actual rephrasing of the program model by using language-agnostic strategies.

Another branch of language-independent transformation specification is based on an XML-based program representation. RefaX [20] brings the premise of a fully language- and model-independent refactoring tool by using XML, while Jrbx [19] generalizes this idea by adding fairly generic static semantic analysis. We share the aim of these approaches, but we add verifiability of transformations at reasonable cost as an additional primary requirement.

Specific languages for refactoring. Designing domain specific languages for refactoring programming is also an established 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.

In the former case, when the refactoring operates on the level of the program model, the actual program representation highly determines the abstraction level of the patterns and the transformation primitives. Gómez et al. [10] introduce a generic model for representing programs and their history, pushing the boundaries of regular program representations in order to support a wide variety of language processing methods.

Rewrite-based transformation languages use different kinds of patterns to match and construct program models. Leitão [16] gives an executable, rewrite-based refactoring language with expressive patterns, Verbaere et al. [28] propose a compact, representation-level formalism for executable definitions. These formalisms are expressive and language-independent, but at this level of generality, correctness checks for refactoring definitions would become practically unfeasible. Rich and high-level program patterns are presented by Gil et al. [9] in their language for defining programming idioms in Java.

Languages for composing already existing transformations exist as well: Kniesel and Koch [12] put the emphasis on ensuring correct composition of transformations, and for Erlang, Li and Thompson [17] define an API for describing prime refactorings and a feature-rich language for interaction-aware composition.

Proven-correct refactoring. For the object-oriented paradigm, Schaefer and de Moor introduce a system [26] in which they reason about semi-formal definitions of a set of basic refactorings. This is a very influencing piece of work, but they focus on preserving static semantic properties, not dynamics. Roberts [23] applies a different definition style, with an emphasis on the side-conditions and proper composition of the base refactorings. Neither of them provides formally verified and executable definitions.

There are some results [27] in defining provably correct refactorings for simple languages, and also some mechanised proofs exist even for modern laguages and real-world use cases [6, 25]; on the other hand, these are specific to one particular transformation, and do not allow for defining custom transformations or provide verification for those. [8] presents a preliminary work on defining verifiable and executable refactoring in Maude, with a similar approach to ours as to rewriting-based definitions, but their definitions are very low-level and hardly readable, out of reach for the average programmer to specify their own transformations.

3 Refactoring definition

Before describing the refactoring framework itself, we elaborate on the abstractions to be used for defining refactoring program transformations. We start by separating concerns. Analysis, condition checking and transformation are seemingly interdependent phases of the refactoring process, but by careful separation we can make the definition less error-prone [18] and enable a more modular implementation. Then, we introduce the refactoring definition abstractions that allow us to define transformations in a high-level and compositional way and enable semi-automatic formal verification for consistency. We end this section by exploring the consequences of defining the transformation with the proposed abstractions, and analyse how it loosens and simplifies the dependencies among the various components of the refactoring framework. This refined dependency structure lets the language-independent components be parametrised with language-specific artefacts and will lead us to a language-parametric framework.

3.1 The refactoring business logic

Refactoring implementations define their input and output as program text in concrete language syntax; however, under the hood, the text is usually turned into an intermediate representation (model) which is further analysed and transformed, and finally, the model is printed back to text. If we properly separate these phases, we can define the refactoring as a composition of *analysis*, *transformation* and *synthesis*.

$$Text \xrightarrow{analysis} Model \xrightarrow{transformation} Model \xrightarrow{synthesis} Text$$

The model we talk about here is a high-level program representation, such as an abstract syntax tree (AST), a higher-order abstract syntax tree or an abstract semantic graph (ASG). In this model, we do not suppose the program logic or high-level architecture to be present (like a UML model) — it is more like a graph that captures the grammatical structure and possibly the static semantic properties of the transformed program. The level of detail in this graph model may vary, we address this question in the next section.

Let us define what we mean by the phases of refactoring:

- *Analysis* is the process of extracting the information from the textual format that is necessary for checking the refactoring side-conditions and locating program elements to be changed by the transformation.

  Analysis can be further divided into two steps: syntactic analysis (parsing) and semantic analysis. Parsing yields a structure tree for the text, then static semantic analysis computes an approximation of fundamental semantic properties of program entities, such as binding relations, data-flow and control-flow.

- The *model transformation is the actual business logic of the refactoring.* It takes place in the middle of the process, given as a (deterministic) relation that maps program models to equivalent program models.

  $$\mathit{Transformation} : \mathit{Model} \to \mathit{Model}$$

  Typically, this function is defined as the semantics of an algorithm written in a programming language or a description in a domain specific language, which does traversals on the model to gather semantic information and carry out rewriting.

- *Synthesis* turns the model back to textual format and obtains the result of the refactoring. We suppose that the model contains the original layout and names of the program and it can be pretty-printed to concrete syntax, but we note that for some higher-level models it may be necessary for the synthesis to incorporate the original text as well.

The strict separation of analysis, model transformation and synthesis simplifies the definition and verification of refactoring transformations, yet the composition of these steps can precisely define the original text transformation. Refactorings may have context-sensitive side-conditions requiring thorough inspection of static semantic properties and therefore complex semantic analysis, but in the rest of the paper, we focus on the model transformation phase.

In particular, from now on by *refactoring* we mean the model transformation and not the text transformation. In the refactoring definition we omit the collection or extraction of static semantic information, and the formal verification of our refactoring definition only proves the model transformation correct, not the text transformation – analysis and synthesis are trusted components in the system.

Level of model abstraction

The complexity of the refactoring definition highly depends on the program model, and the abstraction level of the model affects the complexity of the analysis as well. As it will be demonstrated, the boundary between analysis and transformation is movable by adjusting the level of detail in the model. In this sense, analysis is not the process of building the program model but the extraction of static semantics, which may happen alongside the transformation. Apparently, the more detailed the model is, the less analysis-related traversal takes place during transformation.

Simple model. If the model does not contain details on the static semantics of the program, the transformation gets more complex as it has to carry out analysis tasks. In a corner case, the analysis only does parsing; thus, the program model is a syntax tree and the transformation has to conduct semantic analysis (syntax tree traversals) for checking the side-conditions of the refactoring (see Figure 1). In this case, transformation definitions are overly complex and they are out of reach when it comes to formal verification of semantics-preservation, or even to check termination properties of analysis and transformation. For instance, in the optimisation steps discussed in [3], traversal strategies carry out behaviour-preserving transformations by linking analysis and term rewriting, and the drawbacks of this approach are discussed in [18].

$$\mathit{Text} \xrightarrow{\mathit{parsing}} \mathit{AST} \xrightarrow{\mathit{analysis+transform.}} \ldots \xrightarrow{\mathit{analysis+transform.}} \mathit{AST} \xrightarrow{\mathit{deparsing}} \mathit{Text}$$

Figure 1: Refactoring with tree rewriting

Detailed model. A complex enough static analysis can build a model detailed enough that enables the transformation to check the side conditions by simple queries in the model. This is a trade-off: a more complex static semantic analysis may be harder to reason about, but cuts the complexity from the transformation, making both of them tractable. In the corner case, the model can be so detailed so that predicates in side-conditions are one-to-one mappings to model elements and no analysis-related operations take place during transformation (see Figure 2).

$$\mathit{Text} \xrightarrow{\mathit{parsing}} \mathit{AST} \xrightarrow{\mathit{analysis}} \mathit{ASG} \xrightarrow{\mathit{transform.}} \ldots \xrightarrow{\mathit{transform.}} \mathit{ASG} \xrightarrow{\mathit{synthesis}} \mathit{Text}$$

Figure 2: Refactoring with graph rewriting

Since our main goal with this refactoring framework is to make definitions generic and trustworthy, we aim at using a high-level and detailed program model. This model extends the syntax tree with information on binding, types, data-flow and

control-flow, and even on purity and non-functional properties, so that the refactoring definition side-conditions can be expressed in terms of concepts of the programming language we refactor, and more importantly, analysis and transformation concerns are fully separated. In the case of the Erlang prototype implementation of the proposed framework (see Section 5.2), we rely on the semantic program model introduced in [2].

Tree rewriting in the graph. According to Figure 2, the refactoring transformation is a mapping from semantic graphs to semantic graphs, which suggests that it is easiest defined with a graph rewrite system. Nonetheless, the figure also suggests that any AST can be mapped to the corresponding ASG with static semantic analysis. In practice, the ASG is a proper extension of the AST, containing additional edges and nodes that represent static semantic information.

In our approach we divide the model into its syntactic and semantic parts and work with the model like this: we carry out a transformation on the syntax tree whilst using the semantic layer for checking transformation validity. The result of the transformation is semantic graph containing no semantic elements, so it is reanalysed to obtain the semantic graph prior to further transformation. Since both the AST and the ASG are understood as properly formed models, we can refine the previous signature for transformations discussed in the beginning of this section:

$$\mathit{Transformation} : \mathit{ASG} \to \mathit{AST}$$

Note that although this approach alternates analysis and transformation (see Figure 3), it keeps these phases completely separated (unlike in Figure 1), so still realise separation of concerns . Implementing the transformation as a function from graphs to trees provides a good strategy towards defining trustworthy refactoring in terms of semantics-constrained term rewrite rules for which we introduce abstractions in the following section.

$$\ldots \quad \mathit{AST} \xrightarrow{\mathit{analysis}} \mathit{ASG} \xrightarrow{\mathit{transformation}} \mathit{AST} \xrightarrow{\mathit{analysis}} \mathit{ASG} \xrightarrow{\mathit{transformation}} \quad \ldots$$

Figure 3: Refactoring with semantics-constrained tree rewriting

Summary of assumptions on the model. In the rest of the paper, we assume that the *refactoring is defined on a high-level model* that captures program syntax and static semantics. The *transformation on this model maps the tree along with the static semantic properties into a transformed tree.* We will also assume that the *model's semantic layer captures all program properties that may be needed to tell side-conditions of refactorings* (e.g. name references, data-flow relations or purity of expressions). These target language-level concepts are defined by a set of so-called semantic predicates, which will be used in the side-conditions of refactorings.

3.2 Abstractions for defining transformations

This section surveys how refactorings in the proposed framework are defined such that they provide *trustworthiness* and enable *genericity*. These goals are mainly achieved by keeping the refactoring definition high-level (independent of the representation and the target language), declarative (expressing what to do rather than how to do) and compositional (small definitions combined into bigger ones). We review the refactoring definition abstractions proposed in our previous work [11], and at the same time, we investigate how these abstractions allows for a generic and modular implementation of interpretation and verification.

First of all, higher abstraction level in the definition means less details mentioned explicitly, which reduces the complexity of the definition and the potential for making mistakes. This may come with a performance penalty, but assuming that trustworthiness is more important than efficiency, it is reasonable to opt for higher-level abstractions (as opposed to low-level transformation primitives). For instance, in the context of refactoring, using term rewriting is clearly safer than direct manipulation of the syntax tree as it excludes the risk of constructing structurally invalid subtrees and therefore creating an invalid model.

Careful selection of the transformation abstractions can also make the definitions more amenable to formal verification, further increasing trustworthiness: for instance, refactoring schemes allow us to argue about the correctness of the program transformations in terms of verifying a set of program patterns for semantics equivalence instead of proving imperative term rewrite algorithms correct. This is similar to composing imperative programs with algorithmic skeletons that correctly implement compound control patterns and enable programmers to write complex programs without mentioning the low-level details. Again, this comes with a reasonable penalty: not all program transformations will be expressible with this set of abstractions, but the goal is to be able to define meaningful refactorings in a way that allows for semi-automatic formal verification.

Last but not least, the high-level definitions can be given in a language that does not depend on the representation of programs nor on the particularities of the target programming language, enabling a fairly generic implementation parametrised with language-specific components. The resulting modular framework showcases reusable, language-independent components, as well as it provides trustworthiness by reducing the complexity of individual components (see details in Section 4).

Strategic term rewriting with semantic predicates

The transformation function over models could be defined imperatively, but we aim at defining it as declaratively as possible — as mentioned above, declarative programs contain less details as to how the execution takes place and thus they tend to be more reliable. As one of the building blocks, we employ conditional term rewrite rules to define local tree transformations. This formalism abstracts over the imperative steps of term traversal, pattern matching and replacement construction, serving as a declarative description of simple rewrite steps.

Conditional term rewrite rules consist of two patterns and a condition expression:

$$\frac{\textit{matching pattern}}{\textit{replacement pattern}}\ \textit{conditions}$$

In such a rule, the matching and replacement patterns are first-order terms: they can contain metavariables to extract subterms and use those to construct new terms. In the typical formalisation, the condition is a statement on the rewrite relation itself, potentially referring to the metavariables bound via pattern matching. The set of rules can be interpreted as a normalising term rewrite system by assuming exhaustive application of rules.

Strategic term rewriting improves on ordinary systems by introducing explicit control and context for rewrite rule applications, which is more suitable for defining refactoring transformations. We will use a generalised variant of strategic term rewriting to define transformations over the syntactic part of the semantic program model. In order to accommodate the principle of separation of concerns explained in the previous section, we generalise strategic term rewriting in several aspects: we define conditions in terms of a logic formula using language-specific *semantic predicates (metatheory)*, as well as we introduce strategies that use static semantic properties to control the term rewrite rule application. This latter idea of semantics-driven strategies, so-called *refactoring schemes*, provides fully declarative definition for extensive program transformations as it hides the rule application control under generic control schemes.

Semantic conditions. As mentioned already, [18] gives in-depth explanation of how difficult it may be to reason about side-conditions expressed in terms of reachability statements. To overcome this issue, unlike traditional term rewriting, we do not refer to the rewrite relation in the condition; instead, the conditions are logic formulae over a predicate set characterising the abstractions of the object language. With this design decision, we fully detach analysis and transformation in the refactoring definition, which will allow for a generic implementation in the framework.

Semantic predicates in our approach have two interpretations: they can be evaluated over a particular program model, or can be mapped to a set of axioms in the dynamic semantics of the target language. This latter is of great importance from the verification point of view. For instance, the predicate *pure* can be evaluated by checking the expression for any side-effects, while the axiomatic specification tells that such an expression can be moved in the control chain while preserving control and data flow (for example usage, see Listing 3).

Refactoring schemes. In general, single conditional rewrite rules can only define local changes, so-called *local refactorings*. On the other hand, many refactorings span over entire projects and are inherently *extensive*: they affect many locations in the program, which have to be modified consistently. Strategic term rewriting offers simple operations [29] for combining simple (or local) transformations

with e.g. sequential composition, branching or fixed-point operation, but from the trustworthiness point of view, these combinators are too permissive and tedious to formally tackle. Refactoring is a very special case of program transformation, which gives rise to the idea of strategies specific to program refactoring. We call these *refactoring schemes*.

Schemes are special strategies that combine conditional rewrite rules, and are defined using ordinary control strategies as well as target modifying strategies combined with semantic predicates. They provide a high-level notation for extensive changes, hiding the control primitives and providing a surface language for defining consistent program transformations in a declarative way. Again, schemes pose a restriction on the sorts of expressible transformations, but dividing the definition to multiple levels will allow us to implement the execution modularly and carry out semi-automatic formal verification.

Consistency. The key concept behind schemes is dependency: extensive transformations have to follow dependency chains in the program, visit and change those program elements consistently that are interdependent. Schemes can be driven by dependencies induced by data flow or name binding. Correctness proof for a scheme is as hard as proving an imperative strategy correct; however, the method divides the proof in half: verification of the scheme and the verification of the instantiation. In our design, the second half can be carried out semi-automatically as it boils down to machine-checkable expression pattern equivalences.

Dependencies vary from language to language; hence, our method supposes that the set of pre-verified refactoring schemes are defined for each object language the framework is instantiated for, based on the static semantics of the language. Consequently, although the idea of schemes is language-independent, the concrete schemes we define the transformations with are specific to the object language. In the following section, schemes will be identified as artefacts attached to the object language definition.

Refactoring compositionality

The abstractions for defining local and scheme-based extensive refactorings are supposed to be micro-refactorings: they carry out the least possible amount of transformation steps which form a consistent change in the program. The smaller the steps, the more trustworthy and more easily verifiable they are. Once such a micro-step is proven to be semantics-preserving, it can be combined with other refactoring steps, and the result will be another, compound refactoring.

Our specification language facilitates a sub-language specific to combining refactoring transformations. The steps can be combined by means of basic imperative control: sequencing, branching and (bounded) iteration. These combinators are compositional: if the steps they combine are behaviour-preserving, the resulting transformation will be behaviour-preserving as well. In the end, complex refactorings are defined by decomposition to smaller refactoring steps that are defined as instances of refactoring schemes.

Examples of refactoring definitions

To facilitate the refactoring definition abstractions introduced in this section, we propose a domain-specific language (DSL) [13] for refactoring. Definitions in the refactoring specification language are both executable (can be mapped to a computable model transformation function) and are semi-automatically verifiable (behaviour preservation can be formally checked by verification of automatically synthesized formulae).

Syntax rewrite rules are written in the inference rule notation, patterns are expressed in the concrete syntax of the object language. In the patterns, metavariables are distinguished from ordinary variables by using a kind of quotation syntax (e.g. `#varname`) or extra predicates (e.g. `is_var(VarName)`). In the Erlang prototype, the normal variable syntax is used for metavariables and target language variables are matched with conditions. Metavariables followed by double-dot match consecutive syntactic elements. Schemes are instantiated with rewrite rules, and refactorings are combined in simple scripts. We showcase some examples borrowed from [11], defined for Erlang [4] as the object language.

Local refactoring. Simple, local changes are expressed with conditional term rewrite rules, where conditions are first-order logic formulae constructed with semantic predicates defined by the language metatheory. Listing 1 defines a transformation that encloses an expression into a lambda-abstraction, supposing that the expression does not bind any variables that are referred to by its context (predicate `non_bind`). This definition also demonstrates the usage of matching conditions [29]: the list of free variables (`free_vars`) is bound to a metavariable (`Vars..`) and is being used in the replacement pattern.

```
local refactoring wrap()
    E
    --------------------------------
    (fun(Vars..) -> E end) (Vars..)
when
    Vars.. = free_vars(E) and non_bind(E)
```

Listing 1: Wrap expression refactoring

Extensive refactoring. Extensive changes are defined as reductions to refactoring schemes. For instance, a scheme for Erlang is *function refactoring*, which takes two rewrite rules and visits the definition and the references of the function. References may include intra-module and inter-module function applications, both first-order and higher-order. The patterns given in the parameter rewrite rules define the way the dependent program parts are changed. For the scheme instance to be correct, the two parameter rewrite rules have to be consistent.

The function refactoring scheme can be used for implementing various refactoring steps: renaming a function (Listing 2), reordering and grouping its arguments.

```
function refactoring rename_function(NewName)
definition
    Name(Args..) -> Body..
    -------------------------
    NewName(Args..) -> Body..
reference
    Name(Args2..)
    ---------------
    NewName(Args2..)
```

Listing 2: Rename function refactoring

Interestingly enough, the very same scheme can be used to move a binding from the function body to its signature, introducing a new parameter to the function. In this latter case (see Listing 3), the scheme instantiation contains an extra condition expressing that the expression moved from the body to the call site is pure (does not cause any side effects) and closed (does not contain any free variables).

```
function refactoring var_to_param(X)
definition
    Name(Args..) -> X = E, Body..
    ----------------------------
    Name(Args.., X) -> Body..
reference
    Name(Args2..)
    ----------------
    Name(Args2.., E)
when pure(E) and closed(E)
```

Listing 3: Top-level local variable to function parameter refactoring

Composite refactoring. Let us consider a simple refactoring that lifts a local variable from the function body to the function scope, as a new parameter. This transformation can be decomposed to 1) iteratively lifting the variable to outer scopes (`outer_variable`) and 2) adding it as a parameter (`var_to_param`) once it is a top-level variable. This composition of refactorings can be expressed by iteration and sequential composition, as scripted on Listing 4 (the pseudovariable `This` refers to the object language variable that was selected as the target of the transformation). Note that in these composite refactorings, transformations are applied to program elements determined by so-called selector functions; in this example, `function` selects the enclosing function of the variable originally chosen as refactoring target.

```
refactoring to_function_parameter()
do
  iterate This.outer_variable()
  function(This).var_to_param(This)
```

Listing 4: Local variable to function parameter refactoring

Dependencies untangled

A refactoring framework is said to be language-generic if it is either language-parametric or easily adapted to different programming languages. This aim is highly supported by the abstractions with which we express the implemented refactorings. In this section, we have explained the assumptions we make on the refactoring definition and the abstractions we use for specifying transformations. In particular, term rewriting lets us abstract over tree manipulation, semantic predicates and conditions let us separate analysis from transformation, whilst refactoring schemes serve as another parameter to a generic yet trustworthy implementation.

The refactoring specification formalism, apart from the syntax of the patterns in the rewrite rules, is independent of the object language (we note that although predicates are language-dependent, the condition language over predicate symbols is language-independent). We achieve this language-independence by untangling the dependencies among parsing, semantic analysis, condition checking, transformation and synthesis, and by incorporating the idea of algorithmic skeletons into refactoring. With this, we can decouple the language-independent parts from the language-specific elements, and we can define the latter as plug-in components. In particular, the object language is injected into the framework in terms of definitions for syntax (context-free grammar with metavariable format), static semantics (axiomatic semantic predicates), dynamic semantics (small-step rules) and schemes (semantics-directed strategies).

4 Refactoring framework

The previous section described the abstractions we can use for specifying refactoring transformations in a rather generic way. The proposed specification language is independent of the object language[1], and it can be interpreted in a generic framework parametrized by the definition of the object language. In this section, first we overview the artefacts that parametrise the framework for a particular programming language, then we introduce the components that implement execution and verification of refactoring definitions.

4.1 The object language definition

The following artefacts define the programming language whose sentences are to be refactored. Essentially, they provide a formal definition of the language in question, as well as they define the refactoring idioms that transform program entities according to their control and data dependencies in the object language.

[1] Although if we use concrete syntax in term rewrite rules, the syntax definition of the object language is needed for parsing first-order terms.

Context-free syntax

Parsing and deparsing of both input programs and program patterns in rewrite rules can be driven by a single *context-free grammar* definition, with pattern parsing also guided by the definition of metavariable format [15]. The definition of the object language syntax can be given in the usual BNF-like notation, and bottom-up parsers can be generated from it. In addition, it has to contain the abstract syntax description since the internal representation of both programs and program patterns is based on the AST.

Static semantics (metatheory)

The metatheory is defined in terms of a *set of decidable semantic predicates* supplied *with two different interpretations* (or semantics if you like):

- Evaluation of the predicate on a given model, yielding true or false. This part of the definition is used in the execution components of the framework, in particular the condition evaluation directly refers to the predicate evaluation defined in the metatheory.

- Axiomatic specification in terms of semantic rules which can be used when arguing about semantic equivalence. This part of the definition is used by the verification component in the framework: when proving conditional pattern equivalence, the semantic conditions are mapped to a set of hypotheses in terms of small-step semantic rules.

It is apparent that these two interpretations of the predicates need to be consistent: every time a predicate evaluates to true, the hypotheses on the dynamic semantic have to be valid. This can be checked with respect to the dynamic semantics definition of the language, but the details of this problem are out of the scope of this paper.

Dynamic semantics

The semantics artefact for the object language contains two definitions: the *specification of program behaviour and the characterisation of semantic equivalence* between programs, program fragments or program configurations.

The framework is designed to facilitate a small-step operational-style definition of semantics. During verification, the semantics rules can be used for symbolic rewriting of program patterns checked for semantic equivalence [5]. Recall that neither the semantics nor the behavioural equivalence is incorporated in the refactoring side-condition specification; thus, the execution part of the framework is independent of the dynamic semantics of the object language.

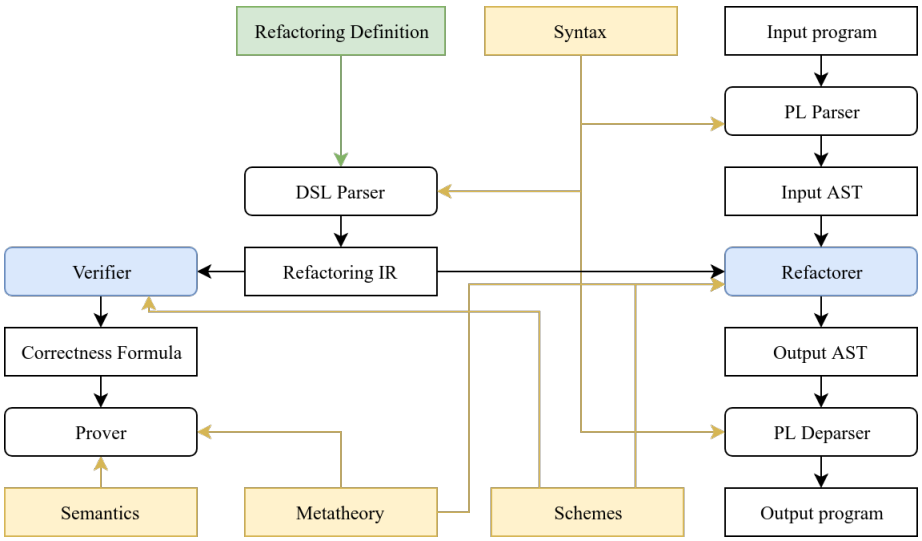

Figure 4: Components and artefacts in the refactoring framework

Refactoring schemes

Schemes are language-specific *refactoring idioms (or transformation templates)*, which are parametrised by conditional term rewrite rules. Like for semantic predicates, for schemes we need to provide two interpretations (or semantics) in their definition:

- From the operational point of view, schemes are transformation skeletons, which can be expanded to strategic term rewritings. Thus, it has to be defined how the declaratively specified scheme is expressed in an imperative strategy that applies rewrite rules at appropriate locations in the program model.

- From the verification point of view, schemes need to be mapped to logic formulae that express the correctness property of the extensive refactoring described with them. Our proof-of-concept implementation translates scheme instances to a set of conditional equivalence formulas over the dynamic semantics of the language.

4.2 Framework components

As discussed above, the framework (see Figure 4) is parametrised by the definition of the object language given with carefully designed abstractions. Once the framework is tailored to a language, language-specific transformations can be checked for correctness or can be run. In particular, the *refactoring definition* (given as

a specification in the formalism discussed in Section 3.2) determines a semantics-constrained term rewriting relation, which can either be checked for correctness, or it can be executed on a particular input program by the framework implementation.

The framework accommodates a frontend and two backends for the two purposes. The frontend, using a lightweight analysis of the refactoring definition, creates the intermediate representation (IR) of the refactoring specification, whilst the two backends implement the two different sorts of semantics for the refactoring specification. In principle, the two backends can be used independently: one can verify refactoring definitions without execution, and the other way around, execute a definition without verification. Nevertheless, this latter brings the risk of altering the program behaviour during transformation, but it is not prohibited for the sake of situations where the formal verification of the refactoring is practically not feasible in a certain time limit, but the refactoring has to be executed anyway.

Frontend

The refactoring definition enters the framework via the frontend component. It parses and analyses the definition, and yields an intermediate representation for the transformation specification which can be fed into either the execution or the verification backend. The refactoring specification language, originated from the Erlang prototype, features no static or strong typing; thus, the frontend only does simple sanity checks on the definitions before passing them to one of the backends.

Since the rewrite rules are composed of first-order terms (syntax patterns) written in the concrete syntax of the object language, parsing of the refactoring definition requires parsing of object language syntax patterns. For this, we use the context-free syntax definition of the object language, generalize it to allow metavariables in place of subexpressions and parse the patterns with it into an AST with metavariables. As a result, we obtain a refactoring definition IR in which we embed object language ASTs.

Execution

One way the refactoring IR can be interpreted is application on a given input program. This is implemented by the compound execution backend, which utilises all object language artefacts except dynamic semantics, and consists of the following components (with the last four grouped into 'Refactorer' in Figure 4):

- **Parser / Deparser** *(uses: Syntax)*
  The input to the refactoring interpreter is program source code, which has to be parsed into a syntax tree before transformation, and needs to be turned back into text following the model transformation. Thus, the context-free syntax definition of the object language is fed into the parser/deparser components which implement the text-to-AST and the AST-to-text conversions, respectively.

- **Scheme expander** *(uses: Schemes)*
  As it was discussed in Section 4.1, schemes have two interpretations, one of which is a translation to lower-level strategies. In this sense, the scheme is a program template for refactoring with holes to be filled with rewrite rules. This component instantiates it with the supplied rewrite rule arguments and yields an imperative term rewrite program.

- **Strategy interpreter**
  This component implements the basic strategies such as composition, left-choice, all-top-down and congruence. Furthermore, we support a number of strategies that rely on the applied program model (abstract semantic graph with references, see [2]), such as applying a rewrite rule on a node, or all of its subtrees, by reference. This component also implements metavariable environments: the metavariables bound with pattern matching are shared with subsequent rewrite rules, thus providing per scheme instance namespaces of metavariables.

- **Condition evaluator** *(uses: Metatheory)*
  The semantic conditions of term rewrite rules are given in terms of object language level predicates combined with simple first-order logic operators. This condition language is interpreted by the condition evaluator component, which relies on the evaluation of predicates over the semantic model of the program. The metavariables used in these formulae are stored in an environment, which is populated by pattern matching executed by the rewrite engine.

- **Term rewrite engine**
  The term rewrite engine carries out the program model transformation (in fact, syntax term transformation) based on the matching and replacement patterns present in the rewrite rules. The current prototypes support expressive syntactic patterns, such as metavariables for matching multiple consecutive nodes in the AST and non-linear patterns, but semantic patterns are not available yet. In addition, in some cases we make use of simple abstract syntax patterns that allow for matching seemingly different concrete syntactic terms.

  The rewrite rules are interpreted in the usual match-and-build semantics: the pattern is matched against the target AST node, creates a substitution (binds the metavariables), builds the replacement subtree, and finally the original node is replaced by the newly created one. Between matching and replacement, the condition evaluator component checks the side-conditions, and the semantics of term rewriting is failure-aware: unsuccessful matching or falsified conditions result in failure of the rewrite rule, which indicates that the rewrite rule was not applicable. Failure is propagated in rule combinators.

Verification

Complementing the execution backend, the verifier implements the other semantics to refactoring definitions: statically check their correctness. Ideally, this happens prior-execution, but as discussed before, the framework does not enforce correctness within the execution backend. The verification backend takes the refactoring specification IR, turns it into correctness formulas and verifies their validity. It is implemented in terms of the following two main components:

- **Verifier** *(uses: Schemes)*
  Correctness of refactoring definitions (the property of semantics-preservation) is defined by the validity of a set of logic formulae expressing conditional semantic equivalence of program patterns. The verifier component is responsible for associating the refactoring definition with this set of formulae.

  In our system, refactoring steps are all phrased as a composition of instances of parametrically verified transformation schemes, and these preverified schemes determine how the correctness formula is synthesized for the transformation. For a transformation to be correct, its scheme has to be correct as well as the instantiation has to be correct. The formulae that this component synthesises express the correctness of the instantiation. The latter formulae in many cases can be automatically proven with respect to the definition of dynamic semantics and semantic equivalence [11].

- **Prover** *(uses: Semantics, Metatheory)*

  This component checks the validity of the formulae synthesized by the verifier. The prover builds on the metatheory definition by utilizing the axiomatic definition of the semantic predicates, the pattern equivalence is proven upon the definition of dynamic semantics and the definition of semantic equivalence. Once the prover has validated the formulae produced by the verifier component, the transformation is guaranteed to preserve the semantics of any program when applied to by the execution backend.

5 Proof of concept

In order to demonstrate the applicability of this generic framework architecture, we investigated instantiating it for two highly different languages: Erlang and Java. This means that we prepared the artefacts detailed in the previous section, i.e. with the appropriate formalism we defined the syntax, static and dynamic semantics for these object languages, as well as we determined some language-specific refactoring schemes with which we can express meaningful refactoring transformations. In this section, we overview the challenges of instantiating the framework for programming languages in general, and for Erlang and Java in particular.

5.1 Parametrising the framework

Beside providing a formal definition of the object programming language from syntax to semantics, instantiation needs the identification of recurring transformation patterns and understanding of dependencies induced among program elements. Ideally, when parametrising the framework for yet another language, the formal definition already exists (syntax, operational semantics, static semantic analysis), yet it is challenging to find those reusable and verifiable schemes for transformations. In the following sections we discuss our framework and its prototype implementations from this perspective.

Defining the metatheory

The semantic predicates provide a high-level interface for embedding static semantic information about the target language into both schemes and scheme instances. Most notably, the metatheory defines what predicates refactoring preconditions should be built from. Constructing the right metatheory for a specific target language is about finding a characterization of its abstractions suitable for both the execution and verification backend.

Naturally, the chosen characterization must be expressive enough to make the preconditions of schemes and scheme instances specifiable. In addition, its elements should also be computable from the underlying program model while executing a concrete refactoring. When composed correctly, the identified functions and predicates must carry enough information to make verification possible.

A starting point towards an appropriate metatheory can be based on the abstractions of the target language, which, of course, are highly influenced by its paradigm(s). Then, the chosen semantic predicates can be iteratively refined in accordance with the requirements above.

Defining semantic equivalence

The underlying notion of semantic equivalence is probably the most determining aspect of a refactoring. Indeed, it is the basis of both intuitional and formal correctness. An oversimplified definition of equivalence can be as abstract as demanding observed programs to produce the same output for the same input. The problem with this, however, is that it is not concrete enough to be conveniently expressible using a proper metatheory. Therefore we propose to replace the aforementioned definition of equivalence with one of its – more easily specifiable – characterizations, e.g. the conformity of data flow, control flow and binding.

We also have to consider that a refactoring usually transforms only some parts of a program instead of its entirety. Generally, a transformation scope specifying the extent of the modified code can be attached to each refactoring. Our assumption is that rather than using a general notion of equivalence – or one of its characterizations –, it is more intuitive to introduce a stricter, but localized variant for each possible transformation scope. In our framework, transformation scope, and therefore equivalence level, can be matched with refactoring schemes.

Designing language-specific schemes

As mentioned earlier, designing refactoring schemes can be the most challenging part of the framework specialization process. There are several key aspects which should be taken into consideration, e.g. generality, usability, verifiability, etc. Schemes must be general enough to be reusable, but not too general, as that would make their instantiation undesirably difficult. On the other hand, we must aim for schemes which optimally split the verification problem, that is checking the general correctness of a scheme wrt. to a contract concerning the rewrite rules it is parameterized by, and checking whether scheme instances satisfy these contracts.

We propose two iterative methods for scheme construction: top-down and bottom-up. The top-down approach starts from a higher level of abstraction, e.g. the level of language elements, and tries to identify schemes based on possible dependencies between the discussed entities. The basis of the bottom-up direction is a number of complex, desirably representative refactorings of the target language, which are then decomposed to microsteps, from which schemes are obtained by appropriate generalization.

Both methods have their advantages and disadvantages. With the top-down method, schemes are inherently general, but not necessarily usable. On the contrary, schemes obtained with the bottom-up method are usable by definition, but their generality is not guaranteed. In both cases, further refinement iterations are required to mitigate these weaknesses. In the former case, more high-level concepts can be added to the dependency analysis; in the latter, more refactorings can be considered during the generalization.

5.2 Erlang

The first implementation of our refactoring framework was made for the Erlang programming language, via generalization of an analysis and transformation tool [2] implemented in Erlang. This preliminary work had a high influence on how we model the program, split syntax and semantics, and even on the separation principle of analysis and transformation. The analysis and transformation system the Erlang implementation of the framework is built on uses automatic, incremental static analysis to keep the semantic layer consistent with the AST; in fact, our framework implementation makes heavy use of the underlying original transformation system.

Erlang has a fairly simple syntax. The static semantics is mainly about language abstractions (modules, functions, variables, types) and their static semantic properties (such as scopes or purity). Basic schemes for Erlang were constructed upon primary sources of data and control dependencies in the language: function calls, variable binding, data-flow have been identified as schemes of extensive changes. Some case studies have been formalized already in the refactoring specification language with the Erlang-specific schemes, one of those is available in detail in [11].

5.3 Java

The metatheory we constructed for the Java prototype characterizes abstractions from the object-oriented paradigm, most notably inheritance, polymorphism and dynamic binding. Here we used the bottom-up approach to obtain schemes by choosing the *lift segment*[2] refactoring as the base transformation. Its decomposition and generalization resulted in four schemes – local, block, lambda and class – and three equivalence levels – local, block and class. For the verification backend, we used K-Java [1] as operational semantics.

The implementation is built on top of a DSL-engineering framework called Xtext, which comes with Xbase, a reusable, Java-like expression language. By modifying its grammar to resemble Java more closely and to accommodate metavariables, a parser for refactoring definitions could be generated automatically. These definitions are compiled on-the-fly to Java code which uses the refactoring API of the Eclipse IDE. Finally, the translated code is dynamically loaded into the underlying JVM instance and made available from an Eclipse plugin. Our approximation of the metatheory is implemented with the Java Development Tools (JDT).

6 Discussion

The ideas discussed in this paper were inspired by a study on high-level, declarative refactoring definitions for Erlang [11]. The concepts of semantic predicates, refactoring idioms and composition operators all seemed to be language-independent, so we adapted the original idea to Java by re-implementing the entire project with JDT and Xtext. After that, we were certain that the two solutions should share a couple of elements that are fairly language-independent.

Apparently, the existing concepts and implementations had to be redesigned in order to expose and extract those language-independent portions, but we managed to obtain a generic design. Although Section 4 presented a fully language-parametric architecture, the proof of concept implementations for Erlang and Java do not share all these language-independent components yet. Nevertheless, realisation of the generic framework for these two substantially different languages justifies that the concept is viable; it is our long-term plan to implement the Erlang and Java tools as proper instances of the language-generic framework.

Before concluding the paper, we briefly evaluate how, and to what extent, the proposed approach allows for generic and trustworthy implementation of refactoring. We also address the idea of language-independent schemes and discuss work in progress on changing the way verification is built into the process.

6.1 Genericity

The proposed design is language-generic: the refactoring specification language is independent of the object language, as well as the implementation framework is

[2] Refactoring *lift segment* lifts a code segment into the superclass as a newly introduced method.

language-parametric. Namely, when a new language needs to be supported, the framework is instantiated for the particular programming language by providing a formal definition of the language (syntax, static and dynamic semantics) along with refactoring schemes (transformation idioms). The main components of execution and verification are shared between instances for different languages.

How do we achieve this? We sort of rephrase and restructure the usual way of defining and implementing a refactoring, and this rephrasing allows us to cut out and abstract away some language-specific elements. In particular, the high-level program model lets term rewrite rules incorporate semantic predicate conditions, and it allows strategies to control term traversal based on semantic dependencies. This separation of transformation concerns leads to a clear separation of the so-called refactoring business logic, which, on the other hand, can be defined in a declarative and language-independent way. The language-independence of the refactoring specification directly implies that the interpretation of specifications can rely on components parametrised with language-specific artefacts.

Language-independence of refactoring schemes. Language-level refactoring schemes enable high-level description of transformations that respect lower-level dependencies. Parametric verification of schemes involves definition of specific equivalence classes of programs, which in turn imply full semantic equivalence under certain circumstances. Even though schemes seem to be totally language-dependent, we have identified some schemes of schemes: for instance, in many programming languages the abstraction of subroutines exists in some way. Function refactoring in Erlang and method refactoring in Java may be understood as specialisations of a language-independent scheme. Schemes may stem from concepts that are shared among different languages, and in the long term, we plan to investigate the possibility of implementing a set of schemes that are defined in terms of concepts common in various languages.

6.2 Trustworthiness

Trustworthiness comes in many forms, ranging from simplicity, modular implementation or open-source code base with excessive testing. In our paper, we focused on enabling semi-automatic formal verification for behaviour-preservation in semantic program model transformations. We managed to split the transformation definition to traversal control and actual term rewriting in a semantics-driven way, which in turn allows these two parts be verified separately, with the latter done semi-automatically.

How do we prove transformations correct? Local refactorings are fairly simple to check. Since these are composed of one single rewrite rule, the correctness is expressed as one conditional equivalence statement of two program or expression patterns. If, under the side-conditions, the rewriting preserves the semantics, the transformation is correct.

For changes that span over multiple expressions, subroutines or even modules, a notion of "completeness" and "consistency" is needed. Namely, the rewriting has to visit all interdependent elements in the program and carry out consistent modifications to preserve the semantics of the entire program. These properties are ensured by the schemes, which provide a declarative means to express complex refactorings. For these extensive changes, we synthesise a set of equivalence formulas that are checked for validity by using the dynamic semantics of the language.

With schemes, we reduce the global equivalence problem to multiple local equivalence problems. This is enabled by decoupling traversal control and actual rewriting. Schemes are verified with respect to some conditions on the rewrite rules they are parametrized with. Verification of schemes requires manual proving; however, having the pre-verified schemes, the instantiation conditions may be automatically checked, making the scheme-based refactoring definitions automatically verifiable. The conditions are equivalence statements on term patterns. Verification of pattern equivalence is not decidable, but in a lot of cases, advanced, problem-specific proof tactics can lead to equivalence proofs. If we express the equivalence formula in reachability logic, there is an algorithm [5] that can be used to determine whether the two patterns can be rewritten to the same form by using rules in the the operational semantics of the language.

Although automatic verification of scheme instances would be a convenient feature from the user's perspective, due to the undecidability of pattern equivalences, in most cases the proving requires some human assistance. We started to redesign the framework such that the object language semantics is formalised in a proof assistant and the pattern equivalence proofs are written by hand. This is fundamentally different from the K framework based solution, but gives more control and opportunities to the user of our system.

7 Conclusion

Refactoring program transformations are essential in large-scale software development for maintaining code quality. Tools that carry out such transformations need to be trustworthy: there has to be an evidence that the program after the refactoring still behaves the same as before. Correctness of the transformation can be checked for each and every application instance, but the ultimate guarantee on correctness is obtained by the static verification of the refactoring definition.

In our previous work, we have investigated refactoring definition abstractions for different object languages, which allow for semi-automatic verification for correctness. In this paper, we have advanced these previous results by generalising our approach over different object languages and designing a unifying refactoring framework. We have shown that the high abstraction level of the definition enables a fine-grained separation of the various components in a refactoring tool, which in turn allows the recognition and extraction of language-dependent elements, leading to a language-generic implementation. Our proposed solution facilitates execution and static verification of refactoring definitions for different object languages.

References

[1] Bogdănaş, Denis and Roşu, Grigore. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015. DOI: 10.1145/2676726.2676982.

[2] Bozó, István, Horpácsi, Dániel, Horváth, Zoltán, Kitlei, Róbert, Kőszegi, Judit, Tejfel, Máté, and Tóth, Melinda. RefactorErl – Source Code Analysis and Refactoring in Erlang. In *Proceedings of SPLST'11*, pages 138–148, Tallin, Estonia, 2011. URL: https://www.researchgate.net/publication/289641474.

[3] Bravenboer, Martin, van Dam, Arthur, Olmos, Karina, and Visser, Eelco. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inf.*, 69(1-2):123–178, July 2005. ISSN: 0169-2968.

[4] Cesarini, Francesco and Thompson, Simon. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009. ISBN: 0-596-51818-8.

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

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

[7] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN: 0-201-48567-2.

[8] Garrido, A. and Meseguer, J. Formal Specification and Verification of Java Refactorings. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 165–174, Sept 2006. DOI: 10.1109/SCAM.2006.16.

[9] Gil, Yossi, Marcovitch, Ori, and Orrú, Matteo. A nano-pattern language for java. *Journal of Computer Languages*, 54:100905, 2019. DOI: 10.1016/j.cola.2019.100905.

[10] Gómez, Verónica Uquillas, Ducasse, Stéphane, and D'Hondt, Theo. Ring: A unifying meta-model and infrastructure for smalltalk source code analysis tools. *Computer Languages, Systems & Structures*, 38(1):44 – 60, 2012. DOI: 10.1016/j.cl.2011.11.001, SMALLTALKS 2010.

[11] Horpácsi, Dániel, Kőszegi, Judit, and Horváth, Zoltán. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. In Lisitsa, Alexei, Nemytykh, Andrei P., and Proietti, Maurizio, editors, Proceedings Fifth International Workshop on *Verification and Program Transformation,* Uppsala, Sweden, 29th April 2017, volume 253 of *Electronic Proceedings in Theoretical Computer Science*, pages 92–108. Open Publishing Association, 2017. DOI: 10.4204/EPTCS.253.8.

[12] Kniesel, Günter and Koch, Helge. Static Composition of Refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, August 2004. DOI: 10.1016/j.scico.2004.03.002.

[13] Kosar, Tomaž, Bohra, Sudev, and Mernik, Marjan. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, 71:77–91, 2016. DOI: 10.1016/j.infsof.2015.11.001.

[14] Lämmel, Ralf. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, RULE '02, pages 15–28, New York, NY, USA, 2002. ACM. DOI: 10.1145/570186.570188.

[15] Lecerf, Jason, Brant, John, Goubier, Thierry, and Ducasse, Stéphane. A Reflexive and Automated Approach to Syntactic Pattern Matching in Code Transformations. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 426–436, 2018. DOI: 10.1109/ICSME.2018.00052.

[16] Leitão, António Menezes. A Formal Pattern Language for Refactoring of Lisp Programs. In *Proceedings of CSMR '02*, pages 186–192, Washington, DC, USA, 2002. IEEE Computer Society. DOI: 10.1109/CSMR.2002.995803.

[17] Li, Huiqing and Thompson, Simon. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Proceedings of FASE'12*, pages 501–515, Berlin, Heidelberg, 2012. Springer-Verlag. DOI: 10.1007/978-3-642-28872-2_34.

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

[19] Maruyama, Katsuhisa and Yamamoto, Shinichiro. Design and Implementation of an Extensible and Modifiable Refactoring Tool. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 195–204. IEEE, 2005. DOI: 10.1109/WPC.2005.17.

[20] Mendonca, Nabor C., Maia, Paulo Henrique M., Fonseca, Leonardo A., and Andrade, Rossana M. C. RefaX: A Refactoring Framework Based on XML. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 147–156, Washington, DC, USA, 2004. IEEE Computer Society. DOI: 10.1109/ICSM.2004.1357799.

[21] Opdyke, William F. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645. URI: http://hdl.handle.net/2142/72072.

[22] Roberts, Don, Brant, John, and Johnson, Ralph. A Refactoring Tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, October 1997. DOI: 10.1002/(SICI)1096-9942(1997)3:4⟨253::AID-TAPO3⟩3.3.CO;2-I.

[23] Roberts, Donald Bradley. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois, 1999. URI: http://hdl.handle.net/2142/81948.

[24] Roşu, Grigore, Ştefănescu, Andrei, Ciobâcă, Ştefan, and Moore, Brandon M. One-Path Reachability Logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013. DOI: 10.1109/LICS.2013.42.

[25] Rowe, Reuben N. S., Férée, Hugo, Thompson, Simon J., and Owens, Scott. Characterising renaming within ocaml's module system: theory and implementation. In McKinley, Kathryn S. and Fisher, Kathleen, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 950–965. ACM, 2019. DOI: 10.1145/3314221.3314600.

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

[27] Sultana, Nik and Thompson, Simon. Mechanical Verification of Refactorings. In *Proceedings of PEPM '08*, pages 51–60, New York, NY, USA, 2008. ACM. DOI: 10.1145/1328408.1328417.

[28] Verbaere, Mathieu, Ettinger, Ran, and de Moor, Oege. JunGL: A Scripting Language for Refactoring. In *Proceedings of ICSE '06*, pages 172–181, New York, NY, USA, 2006. ACM. DOI: 10.1145/1134285.1134311.

[29] Visser, Eelco and Benaissa, Zine-El-Abidine. A core language for rewriting. *Electr. Notes Theor. Comput. Sci.*, 15:422–441, 1998. DOI: 10.1016/S1571-0661(05)80027-1.