Acta Cybernetica **25** (2022) 817–846.

Adaptation of a Refactoring DSL for the Object-Oriented Paradigm*

Dávid J. Németh[ab], Dániel Horpácsi[acd], and Máté Tejfel[ae]

Abstract

Many development environments offer refactorings to improve specific properties of software, but we have no guarantees that these transformations indeed preserve the functionality of the source code they are applied on. An existing domain-specific language, currently specialized for Erlang, makes it possible to formalize automatically verifiable refactorings via instantiating predefined transformation schemes with conditional term rewrite rules.

We present a proposal for adapting this language from the functional to the object-oriented programming paradigm, using Java in place of Erlang as a representative. The behavior-preserving property of discussed refactorings is characterized with a multilayered definition of equivalence for Java programs, including the conformity relation of class hierarchies. Based on the decomposition of a complex refactoring rule, we show how new transformation schemes can be identified, along with modifications and extensions of the description language required to accommodate them. Finally, we formally define the chosen base refactoring as a composition of scheme instances.

Keywords: verifiable refactoring, scheme-based refactoring, microrefactoring, program equivalence

1 Introduction

Software development in practice is usually an iterative process. That is, the end product is not the result of a single step, instead it is constructed by iteratively

*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]ELTE Eötvös Loránd University, Budapest, Hungary, and Faculty of Informatics, 3in Research Group, Martonvásár, Hungary

[b]E-mail: ndj@inf.elte.hu, ORCID: 0000-0002-1503-812X

[c]E-mail: daniel-h@elte.hu, ORCID: 0000-0003-0261-0091

[d]Project no. ED18-1-2019-0030 (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.

[e]E-mail: matej@inf.elte.hu, ORCID: 0000-0001-8982-1398

DOI: 10.14232/actacyb.284280

refining an initial prototype. The longest phase in the development lifecycle, maintenance [17], also requires the modification and extension of existing code. Changes made between two iterations can be seen as source-level transformations.

If such a modification preserves the functional behavior of software, it is called a refactoring. These behavior-preserving transformations are mainly used to improve non-functional properties (e.g. maintainability) without altering the meaning of the program [6]. Many development environments offer refactorings, but we have no guarantees that these transformations indeed preserve the functionality of the source code they are applied on. Moreover, in case of safety-critical systems, formal verification of refactorings can also be deemed desirable.

An existing domain-specific language, explained in Section 3, makes it possible to formalize automatically verifiable refactorings via instantiating predefined transformation schemes with conditional term rewrite rules [9, 8]. This language was constructed for refactorings mainly on Erlang and the functional programming paradigm. The goal of our research was to investigate whether this method could be applied to other paradigms and languages. Since one of today's most popular paradigm is OOP, we have chosen it as our target, and selected Java as its representative due to its high-level nature and widespread support.

The main contributions of this paper are the following:

- A multilayered definition of equivalence for Java programs, used to characterize the behavior-preserving property of discussed refactorings.
- A case study which shows how new transformation schemes can be identified from a complex refactoring rule, along with modifications and extensions of the description language required to accommodate them.
- Semantic functions and predicates related to abstractions of the target paradigm, which we use to describe transformation preconditions.

The rest of the paper is structured as follows. As an introduction, Section 2 presents the foundations of our work and also summarizes results related to it. Section 3 gives a brief overview of the adapted refactoring language. In Section 4 we discuss general decision concerns of the adaptation process, including the choice of target language; refinement of equivalence; and methods to synthesize new refactoring schemes. Section 5 presents a case study where we identify new schemes based on the decomposition of a complex refactoring rule; modify and extend the description language to accommodate the new schemes; formally define the chosen base refactoring as a composition of scheme instances; and provide a step-by-step example of their application to a concrete program. Finally, Section 6 lists further research directions and Section 7 concludes.

2 Foundations and Related Work

In this section, we give an overview on our method's foundations by citing publications that described them, and also present related work on approaches aimed at specifying and/or verifying refactorings. Please note that the direct base of this paper, the scheme-based methodology, is discussed separately in Section 3.

2.1 Foundations

Strategic term rewriting. In his dissertation [10], Kalleberg discusses language agnostic methods for source code analysis and transformation using data and function abstraction techniques. From the latter he emphasizes *strategic term rewriting*, which serves as a basis for program transformations defined with the combination of traversal strategies and rewrite rules. He mentions *System S* [20] as an underlying formal model, and also a possible implementation in the form of the *Stratego* [3] language.

Microrefactorings. The concept of *microrefactorings*, grounded by Opdyke in his fundamental work [13], is widely incorporated to discussions addressing verified program transformations. The main idea of this method is to decompose complex refactorings in order to obtain small, atomic transformations which are easier to write, understand and verify. Then, the original refactoring can be reconstructed using these microsteps as its building blocks, potentially resulting in a pre-verified transformation. Note that in this case, the composite correctness is a consequence of the microsteps being refactorings themselves.

Object-oriented refactorings. In his previously cited publication [13], Opdyke also presents *refactorings characteristic of object-oriented systems*. Based on the method of microsteps, he gives decompositions for three complex transformations, discussing both breakdown strategy and target language metatheory in detail. He also provides informal correctness proofs along with the described refactorings.

Characterization of program equivalence. Schäfer et al. enhance the concept of microrefactorings in two novel aspects [15]. In order to eliminate the need for complex and fragile preconditions, they chose to dynamically check whether the application of a transformation results in an equivalent program. In the cited paper *equivalence is characterized* with the preservation of data flow, control flow and binding. Additionally, intermediate steps in a composite refactoring are allowed to consume and produce code over an extension of the target language, potentially increasing the expressiveness of transformations.

2.2 Related Work

Verbaere et al. present a scripting language for refactoring in [19]. It is a hybrid-paradigm DSL with a functional part for defining transformations and sublanguages based on logic and path queries to describe complex relations between program elements. Compared to our approach, transformations are expressed not declaratively with syntactic patterns, but with imperative commands to modify code at the level of its internal representation. In addition, neither scheme-like language elements, nor verifiability is addressed in the work.

In [11], Leitão proposes a pattern language for refactoring Lisp programs. Like ours, it is a high-level DSL utilizing code patterns with metavariables, therefore transformations are specifiable in it without the knowledge of internal representations. The DSL itself is expressed within Lisp, resulting in an embedding that makes its language elements more easily executable, but at the cost of them containing more syntactic noise. Again, possible transformations are not outlined with generalized strategies, and verifiability is not discussed.

Li and Thompson describe the refactoring DSL of Wrangler, a tool for the interactive and extensible analysis and transformation of Erlang programs in [12]. Like us, they also distinguish primitive and composite refactorings, providing two high-level sublanguages with templates (code patterns) and combinators. The predefined strategies they offer for composite transformations make their descriptions more concise, but they do not apply this methodology to primitive refactorings. Moreover, as the proposed DSL follows the syntax of Erlang closely, definitions contain much syntactic noise.

In [16], Schäfer and de Moor present a high-level yet precise specification language for refactoring definitions. As already mentioned while discussing [15] in the previous section, they still aim for dynamic correctness guarantees instead of relying on overly complex preconditions. The language itself is built upon the abstractions of the target language, but its definitions are still imperative and lack syntactic patterns as well as general strategies.

Garrido and Meseguer present a mathematically rigorous framework for the formal specification and implementation of Java refactorings in [7], which they demonstrate by several verified refactorings. Although the proposed language contains reusable constructs akin to transformation schemes, in general it is defined on a lower level of abstraction than ours: refactorings are specified imperatively in the realm of the underlying formal semantics.

That is, neither of the above-mentioned related works offer a standalone refactoring language that enables users to describe executable transformations declaratively by high-level code patterns, while aiding usability and potential verifiability with general refactoring schemes for pragmatically composable microrefactorings.

3 Scheme-Based Refactoring

The basis of our work is an existing domain-specific language which makes it possible to define executable and verifiable refactorings using syntactic code patterns over the to-be-refactored language [9]. This allows to specify transformations without knowing any internal representations. The main idea of this existing method is to provide pre-verified refactoring skeletons, called schemes, which can be instantiated with conditional term rewrite rules, resulting in composable microrefactorings that serve as building blocks for complex transformations. In the following we give a brief overview of its description language and the verification technique it uses.

3.1 Description Language

The core of the description language is conditional term rewriting – a powerful tool for specifying program transformations based on syntactic patterns. As an illustration of this existing description language, we present a rewrite rule embedded in it. Note how the example resembles Erlang, the original target language of the method we aim to adapt to Java, and OOP in general.

```
    [#Head | #Tail]
    -----------------------
    #X = #Head, [#X | #Tail]
when
    fresh(#X)
```

In this example, the part before the **when** keyword defines the actual transformation in the form of a matching (above the line) and a replacement (below the line) pattern. During pattern matching, corresponding code segments are assigned to matching metavariables (indicated by a hashmark-prefix in the example). The second part (after the **when** keyword) specifies the precondition of the transformation, that is rewriting takes place only if the precondition holds.

The problem with term rewriting, however, is that it is a low-level approach which makes definitions of complex refactorings complicated and error prone, especially in the case of extensive transformations with many compensational modifications (e.g. a refactoring renaming a function has to modify the original call sites as well). To make refactoring definitions safer and even verifiable, the discussed method restricts the set of possible transformations by introducing high-level, reusable refactoring schemes. The provided schemes already contain the necessary control logic, and only have to be parameterized by term rewrite rules to yield concrete microrefactorings.

```
function signature refactoring swapFirstTwoParameters()
    #F(#A, #B, #Ps..)
    -----------------
    #F(#B, #A, #Ps..)
```

The example above, also taken from the original, to-be-adapted language, presents an instance of one of its schemes, namely *function signature refactoring.* The resulting refactoring swaps the first two parameters of the selected function not only in its definition but in its applications as well.

Microrefactorings defined as scheme instances can then be composed to obtain more complex transformations. In the following example, we show how two refactorings can be applied sequentially.

```
composite refactoring f()
do
    g()
    function().h()
```

Note that in addition to (here not explicitly present) combinators controlling the order of application, selectors are also provided to dynamically change the target to be modified. For example, here `h` is executed on the enclosing function of the original target.

3.2 Verification

In this section, we briefly specify the ideas behind, and requirements of, the to-be-adapted method's verification process, which we need to take into account during the adaptation. For a more detailed description, please refer to the original publication [9].

By restricting the set of specifiable transformations, automatic verification becomes feasible. Naturally, the verified property in this case is behavior-preservation with regards to an appropriately constructed definition of equivalence. The chosen formal model is the operational semantics of the target language, which makes it possible to mathematically reason about the execution of programs, e.g. by symbolically computing the possible outputs and side effects of a given function.

The verification process is two-fold. At first, the provided refactoring schemes are manually verified based on assertions concerning their rewrite rule parameters, collectively called the contract of the scheme. Then, scheme instances are examined whether the concrete rewrite rules used in them satisfy the contract of the instantiated scheme. Given schemes are appropriately identified, conformity to contracts becomes automatically verifiable.

Generally, in order to achieve this, contracts should demand no more than equivalences of specific rewrite rule patterns. The reason behind is that in this case, the formal method presented by Ciobaca et al. [4] can be applied to carry out the verification automatically with the correct tooling. The cited work reduces equivalence to the correctness property of a uniquely constructed, aggregated program which becomes verifiable with the formal proof system discussed by Stefanescu et al. [18]. The basis of this method is the operational semantics of the target language, which is embedded into reachability logic [14]. The proposed proof system is sound, but not necessarily complete, however, as neither the to-be-adapted refactoring language, nor our adaptation aims for completeness, the soundness of the verification backend can be considered adequate in both cases.

4 Adapting the Framework

Even though aiming for language independence, the refactoring framework briefly introduced in Section 3 [8] was developed having Erlang as its target language. The main motivation behind this paper is to recreate the existing framework for a significantly different target language, ultimately achieving another step towards making it more language-agnostic. In the following sections we discuss general aspects of the adaptation process.

4.1 Choosing the Target Language

Erlang is a functional and dynamically typed programming language. While selecting the alternative target language, our main concern was to choose a candidate belonging to another paradigm. In this way, we can possibly reason about how the framework should be adapted not only to a different language, but also to a different paradigm. Considering this, it becomes important that the selected language must be as high-level as possible, that is, it should be an appropriate representative of the chosen paradigm without much syntactic or semantic noise.

Due to its popularity, we chose the object-oriented paradigm. As for the representative, we considered classroom-variants of Java, namely COOL [1] and Bantam Java [5] but in the end we decided on Java. The reasoning behind this decision is based on the fact that unlike alternatives listed above, tool support required for executability and formal semantics needed for verification are only available for Java. However, as the main goal of our work is not complete language support, we had to restrict the target language substantially. Moreover, the formal semantics we plan to use defines Java 1.4 [2], therefore we cannot support e.g. generics or lambda functions.

More precisely, our currently supported target language is Java 1.4 – as described by its formal syntax and semantics in [2] – but with the *exclusion* of the following features:

- non-structured control statements, but with the exception of `return` (e.g. no `continue`, `break`, etc.),
- exception handling (e.g. no `throw`, `catch`, etc.),
- modifiers apart from visibility keywords (e.g. no `static`, `final`, etc.),
- field initializer expressions (e.g. no `class A { int x = 0; }`, etc.),
- class initializer blocks (e.g. no `class A { { /* ... */ } }`, etc.),
- local class definitions (e.g. no `class A { class B {} }`, etc.),
- packages, but with the exception of the default one (e.g. no `package a.b;`, etc.),
- reflection (e.g. no `A.class`, etc.),
- concurrency (e.g. no `Thread.start()`, etc.),
- JVM manipulation (e.g. no `ClassLoader.loadClass("A")`, etc.)
- and, naturally, language elements that were introduced in later versions of Java, e.g. generics, lambda functions, annotations, etc. (e.g. no `List<T>`, `(c) -> c + 1`, `@Resource`, etc.).

A number of these restrictions could be bypassed, for example by using anonymous classes instead of lambda functions. Some of them, however, like the loss of generics, indeed limit the current usability of our framework, but we hope to incrementally extend the list of supported language elements in the future.

4.2 Refining Program Equivalence

What transformations we consider refactorings is mainly determined by the underlying notion of semantic equivalence. Indeed, both intuitional and formal correctness is based on its chosen definition, which is also an important parameter of the verification backend discussed in Section 3.2. An oversimplified version of the classic characterization of program equivalence demands observed programs to produce the same output for the same input. The problem with this notion, however, is that it is not close enough to abstractions of the target language for a refactoring programmer to being reasoned about on the level of source code. To overcome this issue, we propose to replace the aforementioned definition of equivalence with one of its – more easily specifiable – characterizations, e.g. the preservation of data flow, control flow and binding, as suggested by Schäfer et al. [15].

In addition, individual refactorings are mainly designed not to modify a whole program, but rather specific parts of it – we call the actual extent of a transformation its *scope*. We claim that as a result, it is more natural to think and reason about the correctness of a refactoring concerning only its scope. To support this assumption, instead of using a global definition of program equivalence, we introduce several, generally stricter variants of it, specialized for the possible types of transformation scopes. Ideally, it must be separately proven that each local equivalence implies the chosen global one. In the discussed framework, transformation scope, and therefore equivalence level, can be matched with refactoring schemes.

The following example shows – possibly in its simplest form – how ambiguous it could be to reason about program equivalence in case of partial code fragments, typically seen in rewrite rules.

```
int x = 0;
```

```
int x = 1;
```

Deciding whether the specific code transformation of rewriting the first variable declaration to the second one should be considered a refactoring is not straightforward. In fact, the answer depends on the wider context: if the modified statement is located in an unused private method, the behavior of the enclosing program is guaranteed to remain the same. However, a situation where the behavior truly changes can easily be imagined. This ambiguity is why we reason about a scope-dependent equivalence definition: we do not want the developer of the refactoring to think about conditions which reach out of the current transformation scope.

Another interesting observation arises from the examination of the following pair of class definitions:

```
class A {
    public int f() { return 6; }
}
```

```
class A {
    public int f() { return 3 * g(); }
    public int g() { return 2; }
}
```

In this case, the first question is that how do we characterize the meaning of a class definition? Based on intuition, our proposal is to reduce their equivalence to the equivalence of their public interfaces. On the other hand, adding a new method to a public API while preserving the semantics of its existing methods surely does not alter the previously accessible functionality of the examined library. Therefore, this example shows that the mathematical relation we are looking for is not necessarily an equivalence ($\equiv$, i.e. a relation which is reflexive, transitive and symmetric): exchanging the property of symmetry for antisymmetry, the resulting partial ordering ($\preccurlyeq$) can model the asymmetric nature of program transformations better than an equivalence.

In conclusion, we summarize three types of equivalence levels:

- **Local.** In the lowest level of abstraction, i.e. in case of a refactoring defined over expressions and statements, we cannot leverage any information about the environment of the target. Therefore, here we expect syntactic equivalence.

- **Block.** One abstraction level higher, in case of refactorings concerning code blocks, we can assume that the overall behavior does not depend on block-local variables as long as the blocks themselves are equivalent. We can extend this level to methods if we consider their bodies blocks and their formal parameters block-local variables.

- **Class.** As mentioned above, in case of refactorings modifying classes we only want to ensure that the transformed class hierarchy provides at least the public interface of the original library, but also in a semantically block-equivalent way.

4.3 Synthesizing Schemes

When trying to tackle the task of constructing new refactoring schemes, we have to take three main design goals into account: generality, usability and verifiability. The first two of them are interconnected, as schemes possessing a high level of generality tend to be more difficult to instantiate; and conversely, schemes usable with minimal effort usually show a lack of generality. The additional requirement of verifiability demands schemes to appropriately split the two-fold correctness-checking problem between proving their parametric validity wrt. their contract, and checking whether concrete rewrite rules in scheme instances satisfy these assumptions.

With the aim of invoking an intuitional understanding in the reader, we present the main ideas behind two possible iterative techniques for scheme construction.

- **Top-down.** The top-down approach starts from a higher level of abstraction and tries to identify new schemes, or concretize (break down) existing ones based on general categorization possibilities. Two recommended initial categorization dimensions could be the elements of the target language and aspects of program equivalence. For example, if our target language offers only fields and methods, and we characterize program equivalence with data flow and binding, the top-down method would yield 4 initial schemes: data flow refactoring of fields, data flow refactoring of methods, binding refactoring of fields, binding refactoring of methods.

- **Bottom-up.** The basis of the bottom-up direction is a number of complex, desirably representative refactorings of the target language. Firstly, these complex refactorings have to be decomposed in order to obtain microsteps from them. Then, the resulting refactorings are generalized until they become schemes. Finally, the results can be validated by reconstructing the original base refactorings from scheme instances. Instead of providing a concrete example here, we refer the reader to Section 5, where we discuss this approach in detail.

Both methods have their advantages and disadvantages. The top-down technique yields general schemes by definition, but usually the results are too abstract to be practically usable. On the contrary, schemes constructed with the bottom-up method are inherently usable, but not necessarily general. We can overcome these weaknesses by refining the obtained schemes iteratively. In the former case this means the consideration of additional categorization possibilities, while in the latter more concrete refactorings can be added as a base of the generalization process.

5 Case Study

In this section, we present a case study where we identify new schemes based on the bottom-up method. That is, we select and decompose a complex refactoring; modify and extend the description language to accommodate the newly generalized schemes; formally define the chosen base refactoring as a composition of scheme instances; and finally illustrate the usage of the described transformations with their step-by-step application to a concrete example.

Please note that we do not consider the chosen refactoring and the resulting scheme instances as main contributions of this paper. They rather provide only a base for the presented adaptation process of the language discussed in [9] and [8] to OOP. Constructing a refined and widely usable scheme library for Java is still a future work of ours – see Section 6 for details.

5.1 The Base Refactoring

As the result of the bottom-up scheme synthesis process highly depends on the chosen base refactoring, it is crucial to select one which can be considered a suitable representative of program transformations defined over the target language. Related work offer numerous candidates. For example, we could pick *generalize function* from [8] or *extract method* from [15]. Although on these we could illustrate the concept of decomposition, none of them depend heavily enough on object-oriented abstractions. On the other hand, refactorings from Opdyke [13] are too general and complicated for our purposes.

To overcome this problem, we specifically construct a refactoring capable of demonstrating both decomposition and object-oriented concepts. For the former, we reuse *extract method* from Schäfer, which we extend with *lift method* found in Opdyke's dissertation to address the latter. We call this construction *lift segment* and informally specify its semantics as follows. This refactoring, when applied to a consequent region of statements (code segment), lifts its target to the superclass in a separate method. The arguments of the transformation are the visibility and name of the method to be introduced. See Figure 1 for a concrete example.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int x = 1;
        a = x;
        g();
        int y;
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {
    int a, b;
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
}
```

Figure 1: Program code before and after *lift segment*. The refactoring was applied to the segment marked in blue on the left, with the *package* visibility modifier and `h` as function name. Parts of the code changed by the transformation are also highlighted in blue on the right.

5.2 Decomposition of the Base Refactoring

To advance towards new schemes, we present the decomposition of the base refactoring step-by-step. We start by dividing the base refactoring itself, and then we continue breaking down the resulting subtransformations recursively, until we get refactorings which are sufficiently simple. For each decomposition step, we provide both an informal reasoning and also a concise list of the derived subtasks. Please note that these descriptions are only meant to invoke a high-level insight in the reader. The presented microrefactorings will be explained in detail in Section 5.7.

As mentioned, first we need to decompose the base refactoring. The joining point between the two main components of our custom construction seems natural to choose for its division.

Lift segment:

1. extract segment,
2. lift method.

The decomposition of *extract segment* has already been presented by Schäfer et al. [15]. Their process consists of three iterative steps, where each of them refines the result of the previous one. This decomposition is specifically defined in a way that separates transformations which together would potentially modify more than one of the three equivalence aspects, namely control flow, data flow and binding.

Extract segment:

1. move segment to block,
2. extract block to lambda,
3. refine and extract lambda.

To preserve name binding, statements of the selected segment is moved in reverse order, one by one to a new block. In this way, variable declarations can be handled separately – this is indeed required, as extracting a referenced declaration might change bindings, and therefore behavior.

Move segment to block:

1. insert new block,
2. move selected statements in the block one by one, in reverse order, handling variable declarations separately.

After the initial segment has been extracted to a new block, it can be transformed into a lambda without the fear of modifying binding during the process. At this point, however, control flow becomes fragile because of the potential jump statements located inside the segment. If value returns are present, the return type of the new method can also be calculated here.

Extract block to lambda:

1. inspect jump statements,
2. inspect external assignments (due to limitations of Java).

In the next step, transformations modifying the data flow are used to make data dependencies related to the lambda (and thus to the initial segment) explicit in the form of parameters and a possible return value. Finally, the now binding- and data/control flow independent lambda can be extracted.

Refine and extract lambda:

1. make data dependencies explicit,
2. extract lambda to method.

As the last step, based on the decomposition of Opdyke [13], we define how a method should be lifted to its superclass.

Lift method:

1. lift referenced local fields,
2. lift referenced local methods[1],
3. lift independent method.

In the following subsections we show how microrefactorings listed above can be defined with the adapted framework.

5.3 Extending the Description Language

In this section we discuss modifications and extensions of the description language required to accommodate the new refactoring schemes. As part of the process, not only do we introduce new scheme clauses related to the object-oriented paradigm, but we also mention new language elements meant to make even existing refactoring definitions more concise.

In the base language, pseudovariable **`this`** represents the target of the current refactoring. Because this identifier has a different meaning in the object-oriented paradigm, we replace ours with **`target`** to avoid confusion. Additionally, we make syntactic patterns more expressive by the flexible handling of the ; delimiter in them: instead of a concrete syntactic element, we interpret it as an abstract sequencing symbol with multiple possible manifestations. For example, pattern `#S;#S'` matches both `{}{}` and `{};{}`.

Finally, we introduce the following scheme clauses:

- **`target`**: A clause dedicated to match the target of the refactoring. On the one hand, this can eliminate pattern duplications in the original matching pattern. Moreover, referring to the context of the target in the matching pattern also becomes possible. The following examples respectively present the above-mentioned interpretations of the target clause:

```
    target
    ----------
    { target }
target
    #S ; #S'
```

```
    target ; #S'
    ------------
    #S' ; target
target
    #S
```

  Concrete usage analogous with the previous examples can be found, respectively, in scheme instances `moveIntoNextBlock` and `moveToTop`, see Section 5.6. The first one eliminates duplication from the description of a move transformation by using the target expression in its rewrite-pattern-pair. The second one uses the target expression to make the selected code's environment accessible for a precondition.

[1]Note the recursion.

- **`shadowed references`**: Clause for specifying a compensational transformation for changes in binding induced by moving code. For example, this is where we can restore binding to a field by using the **`this`** qualifier after shadowing it with a local declaration – as seen in the description of the block refactoring scheme in Section 5.5.

- **`top level definition`**: Clause to define or modify a file-level program entity (e.g. class or interface). For example, the second variant of the lambda scheme uses this clause to modify an interface, as seen in Section 5.5.

- **`definition in class`**: Clause for defining a new member (e.g. field or method) inside the enclosing class of the refactoring target. A concrete example can be found in scheme instance `extract` in Section 5.6, where the clause is used to introduce a new method.

- **`definition in super`**: Clause for defining a new member inside the superclass of the refactoring target's enclosing class. This is, for example, how we lift a method in the second variant of the class refactoring scheme by removing it from the base class and reintroducing it in the superclass by the *definition in super* clause, see in Section 5.5.

5.4 Constructing the Metatheory

By metatheory we denote the semantic functions and predicates that capture, and provide a high-level interface for, various static semantic information about the target language. In particular, the metatheory defines what predicates the preconditions of schemes and scheme instances can be built from. To make the metatheory intuitively usable, we define these functions and predicates over a high-level model that closely resembles the abstractions of the target language. The basis of this model is the AST metamodel, therefore we will implicitly use operations commonly defined on it.

In the following we group elements of our metatheory by the semantic property they provide information about, separately listing the ones which are closely related to the object-oriented paradigm.

Data flow. Information about data flow can be used to check variables and fields before and during a transformation, or even while verifying a scheme or an instance. Here we declare functions for obtaining variables and fields through a given entity: `variableReads`, `variableWrites` and `accessedFields`.

Control flow. One of the main notions of control flow analysis is the concept of the path of execution, describing a possible ordering of statements for a given language entity. Here we reuse the definition of `controlSuccessor` and `exitNode` from Schäfer et al. [15], referring to possible control flow successors and the symbolic exit point of a method, respectively. We also introduce the `callGraph` of a method definition, which is the maximal, directed, vertex-labeled graph of method definitions containing all *possible* (see dynamic binding) call relations starting from the selected method definition.

Binding. One of our most important semantic functions is `definitionScope`.

Definition 1. *The scope of method definition d is the set which contains exactly the classes whose instances resolve method calls with d's signature to d.*

As a demonstration, consider the following example:

```
class A {
    public void f() { /* ... */ }
}
class B extends A {
    public void f() { /* ... */ }
}
class C extends B {
    public void f() { /* ... */ }
}
class D extends B {}
```

Here the scope of `A::f()` consists of `A`, the scope of `B::f()` consist of `B` and `D`, and the scope of `C::f()` consist of `C`.

Paradigm. Statically reasoning about the behavior of object-oriented software is made difficult by dynamic aspects of the paradigm, namely polymorphism and dynamic binding. Apart from trivial query functions (e.g. `isSubType`, `superHierarchy`, `subHierarchy`), our main concern here is to find a proper approximation of behavioral properties which may influence our class-conformity relation (see Section 4.2).

Considering the microrefactorings we identified in Section 5.2, a number of them requires a new method definition to be added into a class. In the following, we will call such to-be-added definitions *predefinitions*. Our goal regarding the metatheory is to provide a safe approximation for deciding whether a predefinition could potentially change how a method reachable from public API behaves. For the sake of simplicity, we do not statically check whether two method definitions are equivalent – we simply assume that they are not. In conclusion, we propose the following definitions about the so-called intra- and interhierarchy-reachability of a predefinition.

Please note that the following definitions are meant as readable alternatives to the underlying logic formulae.

Definition 2. *We say that predefinition p is* reachable *if it*

- *overrides a method,*
- *and is* inter- *or* intrahierarchy-reachable*.*

Definition 3. *We say that predefinition p is* interhierarchy-reachable *if there exists a definition which*

- *is located outside the class hierarchy of p*
- *and refers to a signature of p that*
  - *is qualified with either one of the superclasses of p's enclosing class,*
  - *or with a class belonging to the definition scope of p,*
- *and which is* non-constrained intrahierarchy-reachable.

Definition 4. *We say that predefinition p is* intrahierarchy-reachable *if*

- *it overrides a public method,*
- *or there exists a definition which*
  - *refers to the unqualified signature of p*
  - *and is D_p*-constrained intrahierarchy-reachable *where*
    - *D_p is the definition scope of p.*

Definition 5. *Definition d is D*-constrained intrahierarchy-reachable *if*

- *D_i is not empty, and*
  - *either the visibility of d is public,*
  - *or there exists a definition d' that*
    - *refers to the unqualified signature of d,*
    - *and is D_i*-constrained intrahierarchy-reachable
  - *where D_i is the intersection of D and D_d where*
    - *D_d is the definition scope of d.*

Definition 6. *Definition d is* non-constrained intrahierarchy-reachable *if it is*

- *D_d*-constrained intrahierarchy-reachable *where*
  - *D_d is the definition scope of d.*

In short, interhierarchy-reachability denotes whether a predefinition could be called from an external public API, while intrahierarchy-reachability indicates if a predefinition could be resolved from a public API inside its class hierarchy. The reason why the latter is slightly more complex is the fact that in that case, there is a possibility for specific call-chains, starting from a public method and almost reaching a predefinition, to be broken due to disjoint definition scopes.

We illustrate these reachability-definitions with the following three examples. In the first one, the essence of interhierarchy-reachability is shown.

```
class A {
    protected void f() { /* ... */ }
}
class B extends A {
    /* protected void f() { /* ... */ } */
}
```

```
class C extends B {}
class D extends C {
    protected void f() { /* ... */ }
}
class X {
    public void g(A a, C c, D d) {
        a.f(); c.f(); d.f();
    }
}
```

Here, the now commented-out `B::f()` denotes the to-be-added definition. Its enclosing hierarchy consists of classes `A`, `B`, `C` and `D`, where `B` and `C` form its definition scope. Class `X` lies outside of this hierarchy. In definition `X::g(A, C, D)`, which is obviously reachable because of its public visibility, signatures `A::f()`, `C::f()` and `D::f()` are called. In this method, the `d.f()` call is safe, as the dynamic type of `d` can only be `D`, and `D` is not in the scope of predefinition `B::f()` – therefore, the newly added method could not possibly be called. However, the other two calls are unsafe, because for both of them there exists a compatible class from the scope of the predefinition. For example, in both cases the dynamic type of the called object can be `C`, which would result in signatures `A::f()` and `C::f()` being resolved to predefinition `B::f()` – therefore, it is interhierarchy-reachable.

The second example demonstrates an intrahierarchy-reachable predefinition.

```
class A {
    protected void f() { /* ... */ }
    public void g() { f(); }
}
class B extends A {
    /* protected void f() { /* ... */ } */
}
```

Once again, the commented-out `B::f()` denotes the predefinition. The publicly defined, and therefore intrahierarchy-reachable `A::g()` in its superclass calls `A::f()` without qualifiers. Generally, this would not necessarily be problematic – see the next example. This call, however, is still unsafe, because the definition scopes of `A::g()` – i.e. {`A`, `B`} – and `B::f()` – i.e. {`B`} – are not disjoint. Indeed, as a result, on instances of `B`, the publicly accessible signature `B::g()` is resolved to definition `A::g()`, which then calls predefinition `B::f()` in place of signature `A::f()`.

The last example shows how disjoint definition scopes can prevent a predefinition from being intrahierarchy-reachable.

```
class A {
    protected void f() { /* ... */ }
    public void g() { f(); }
}
class B extends A {
    /* protected void f() { /* ... */ } */
    public void g() {}
}
```

Compared to the previous example, definitions `A::f()` and `A::g()`, as well as predefinition `B::f()` remain the same. The only difference is the introduction of definition `B::g()`, which reduces the definition scope of `A::g()` to just {`A`}. As a result, the scopes of `A::g()` and `B::g()` become disjoint, thus there are no classes where `A::g()` could call `B::f()`. Because there are no other references to signature `f()` inside this hierarchy, the predefinition here is not intrahierarchy-reachable.

5.5 Identifying Refactoring Schemes

During the discussion in previous sections, we introduced all concepts and tools necessary for constructing our own schemes, based on a generalization of microstreps that were presented in the form of the chosen base refactoring's decomposition. To define a scheme, we have to provide its name, potential clauses, rewrite control logic, preconditions and contracts. We also assign a level of equivalence to each scheme in accordance with its transformation scope. In the following we briefly show the four scheme(families) we specified: local, block, lambda and class.

Local refactoring scheme. The local scheme can be used to define simple, block-local refactorings on the level of single expressions and statements. It has no special preconditions or control logic.

```
1  local refactoring <name>
2      <matching pattern>
3      ---------------------
4      <replacement pattern>
5  target
6      <optional target pattern>
7  when
8      isInsideBlock(target)
9      and <optional preconditions>
```

In fact, the local scheme can be considered as a way to purely embed conditional term rewrite rules into the language. The sole precondition requires its target to be inside a block (line 8). Naturally, the assigned equivalence level is *local* and the contract of the scheme demands the matching and replacement patterns to be locally equivalent when preconditions hold.

Block refactoring scheme. The block scheme can be seen as an extension to the local one. It can be used to refactor entire code blocks and even contains some simple control logic.

```
block refactoring <name>
    <matching block-pattern>
    ---------------------------
    <replacement block-pattern>
target
    <optional target pattern>
```

```
 7  shadowed references
 8      #reference
 9      --------------
10      #qualifiedName
11  when
12      #qualifiedName = #reference.qualifiedName()
13      and <optional preconditions>
```

Inside the matching and replacement patterns we allow a special type of pattern matching. If keyword **`target`** is explicitly referenced there in a way that its context is unboundedly matched (this is achievable with multipatterns (e.g. `#S..`) at the beginning and/or end of mentioned pattern holes), bounding multipatterns will be matched until the boundary of the enclosing code block. See instance `moveIntoNextBlock` in the next section for an example.

If we move statements in a block, it is possible to introduce unwanted variable shadowings, therefore to alter the original binding. The control logic of the scheme, as can be seen in its **`shadowed references`** clause, automatically compensates this by appending the original name qualification (line 12) to the shadowed reference.

This scheme was designed with the *block* equivalence level in mind. Its contract requires the block-equivalence of matching and replacement patterns considering preconditions and automatic name qualification.

Lambda refactoring scheme. As Schäfer et al. [15] suggest in their work, lambda functions are practical because of their ability to act either as data or as code, making it possible to destruct error-prone refactorings which modify both data and control flow at once into multiple smaller, cleaner transformations. Unfortunately, the formal semantics we plan to base the verification on does not support lambda functions, therefore we have to use interfaces and anonymous class instances instead.

We have identified two variants of the lambda scheme: one for introducing new and one for modifying existing lambda interfaces and instances. Here we present the latter.

```
lambda refactoring <name>
    <matching lambda-pattern>
    ----------------------------
    <replacement lambda-pattern>
top level definition
    <interface definition (#F) for the matching lambda-pattern>
    ----------------------------------------------------------
    <interface definition for the replacement lambda-pattern>
when
    #F.references().size() = 1
    and #F.references().contains(target)
    and <optional preconditions>
```

The main task here is to automatically update underlying interface definitions. For example, we expect the framework to propagate changes between the matching and replacement lambda applications to the corresponding interface, denoted by metavariable `#F` in the description of the scheme. As we want to keep the transformation scope local (we are only discussing special method calls), in the preconditions we verify that the lambda interface is not used anywhere else, i.e. it is only referenced once (line 10) and that one reference is the target of the refactoring (line 11). In accordance with this, the scheme is based on the *local* level of equivalence and its contract demands the matching and replacement patterns to be locally equivalent considering preconditions and interface versions.

Class refactoring scheme. The class scheme was designed for refactorings that modify classes and class members. We constructed three variants in this category: one for introducing new methods, one for lifting methods and one for lifting fields. The reason behind excessive concretization was mainly the complexity of preconditions: we wanted to hide them from users inside the scheme. In the following we discuss the variant intended to add new methods into the enclosing class.

```
class refactoring <name>
    <matching pattern>
    ------------------
    #name(#args..)
target
    <optional target pattern>
definition in class
    #visibility #type #name(#params..) #body
when
    /* omitted for the sake of readability */
    and <optional preconditions>
```

Structurally, the scheme looks quite simple: a new method is introduced into the enclosing class, and the matching pattern is replaced with a corresponding function call. Missing arguments (e.g. `#name`, `#body`, etc.) must be inferred from concrete instances. However, the scheme's true complexity is encoded into its preconditions which we omitted here for the sake of readability. In short, we have to guarantee that the new definition will not cause compiler errors (names are unique, in case of overrides visibility and types are correct, etc.) and also want to check that the predefinition is not reachable (see Section 5.4).

Naturally, this scheme uses the *class* level of equivalence (which, in this case, is technically a partial ordering ($\preccurlyeq$), see Section 4.2). However, due to the exhaustive preconditions, its contract only requires the matching and replacement patterns to be locally equivalent. Of course when checking this we assume that the preconditions hold and that the new method definition – which is called in the replacement pattern – has been inserted to the enclosing class.

5.6 Defining Scheme Instances

Using the schemes introduced in the previous section, we can define the decomposed microrefactorings as scheme instances. At the end of this part, we also demonstrate how the base refactoring can be rebuilt from microsteps in a composite specification.

Local refactorings. The only local refactoring is the one which appends a new, empty code block after its target statement.

```
local refactoring introduceEmptyBlockAfter()
    #s
    -------
    #s ; {}
```

Block refactorings. There are two block instances: one for moving a statement into its subsequent block (`moveIntoNextBlock`) and one for handling declarations differently during the process (`moveToTop`). Here we show only the first one, but the latter could be easily constructed as well.

```
1 block refactoring moveIntoNextBlock()
2     target ; { #S.. } ; #S'..
3     -------------------------
4     { target ; #S.. } ; #S'..
5 when
6     isSingle(target)
7     and (isVariableDeclaration(target) ->
8         not isReferencedIn(target.declaredVariable(), #S'..))
```

Note how the transformation is expressed using only pattern matching. As we can see from the precondition, the block-matching feature helps to obtain the surrounding context without the use of semantic functions. Here we require the target to be a single statement (line 6) – as we want to move only one statement at a time, see Section 5.2 –, and also if it is a declaration (line 7), its declared variable should not be referenced in the remainder of the block (line 8) – since for these references, the original declaration would become invisible if we moved it into the sub-block.

Lambda refactorings. In total, we have defined three lambda refactorings. One introduces a void (`wrapInVoidLambda`), the other one constructs a value-returning lambda (`wrapInValueLambda`). Now we present the third one (`extractInVariables`), which makes data dependencies of an existing lambda explicit.

```
lambda refactoring extractInVariables()
    new #F() { public #type #name() #body }.#name()
    -----------------------------------------------------------------
    new #F() { public #type #name(#inVars..) #body }.#name(#inVars..)
when
    #inVars.. = #body.variableReads().filter(#read :
            isBefore(#read.variable().declaration(), target))
        .map(#read : #read.variable()).reduce()
```

In the first pattern, metavariable `#F` will be matched to the underlying interface of the targeted "lambda"-application. Here you can also see that metavariables might even be assigned in preconditions. In this particular example, variables read in the body of the target lambda (`#body.variableReads()`), but declared before (not inside) it (line 7), are collected into and later used through metavariable `#inVars`... We also take advantage of the fact that the collected variable names can be used both as formal and actual parameters. (The last line of the precondition is a technicality: we have to convert the filtered variable reads to the read variables, and also eliminate duplications (`reduce()`) from the resulting collection, as it will be used as a parameter/argument list.)

Class refactorings. Out of the three class refactoring instances, we show the most interesting one, that is which extracts a lambda to a new method (`extract`). The other two (one for lifting methods and one for lifting fields, both named `lift`) can be mechanically specified without significant extra content.

```
 1  class refactoring extract(#visibility, #newName)
 2      new #F() { public #type #name(#params..) #body }.#name(#args..)
 3      ------------------------------------------------------------
 4      #newName(#args..)
 5  definition in class
 6      #visibility #type #newName(#params..) #body
 7  when
 8      isSubsetOf(#body.dataAccesses().map(#access : #access.target()),
 9          union(#params.., target.enclosingClass().fields(),
10              #body.localVariables()))
```

Similarly to the previous example, metavariable `#F` in the first pattern will be matched to the underlying interface of the targeted "lambda"-application. The difference here is that we have to check whether the lambda to be extracted is truly independent from its surroundings, that is, it does not reference variable-like entities from outside its parameters, body and accessible fields. In other words, the referenced variables (`#access.target()`) of its body's data accesses (`#body.dataAccesses()`) should form a subset of the union of its parameters, local variables and accessible fields of the enclosing class (line 9-10). This instance also has two parameters, the name and visibility of the new method.

Composite refactorings Finally, we can reconstruct the initial *lift segment* refactoring in the composite definition of `lift`.

```
composite refactoring lift(#visibility, #name)
do
    extract(#visibility, #name)
    #extractedMethod = target.enclosingMethod()
    #extractedMethod.cascadedLift()
when
    isSegment(target)
```

Note that here, `extract` and `cascadedLift` are further composite refactorings built from scheme instances mentioned above. In this specification it is also shown how selectors and combinators can be used to imperatively control the application of transformations: in line 4, we store a reference to the method extracted in the previous line in a metavariable, then in the next line we lift it together with its dependencies by applying `cascadedLift` through the referencing metavariable.

5.7 Example

In this section, the previously discussed scheme instances and composite refactorings are demonstrated by the stepwise transformation of a concrete Java program (see Figure 1). Each step is presented by a code fragment pair showing the before-after state, followed by a short explaination of the applied transformation. Code highlighted in blue denotes the target of the refactoring on the left, and the currently modified parts on the right.

The first step is to apply instance `lift` with the *package* visibility and function name `h` as parameters. Refactoring `lift` is composite – within it, `extract` is called, which is also composite, and its first step is `introduceEmptyBlockAfter`.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int x = 1;
        a = x;
        g();
        int y;
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int x = 1;
        a = x;
        g();
        int y;
        {}
        a = y;
    }
    void g() { a = b = 0; }
}
```

After that, `moveIntoNextBlock` is applied, but it does not succeed, because the scope of a declaration that is used in the enclosing block cannot be reduced. The `moveToTop` rule is tried next, which will be performed successfully.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int x = 1;
        a = x;
        g();
        int y;
        {}
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        g();
        {}
        a = y;
    }
    void g() { a = b = 0; }
}
```

The next statement is moved to the target block using `moveIntoNextBlock`.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        g();
        {}
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        {
            g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

We move the first statement of the originally selected segment to a block using `moveIntoNextBlock` as well.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        a = x;
        {
            g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y;
        int x = 1;
        {
            a = x;
            g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

Since the resulting block does not contain a `return` statement, we can use the `wrapInVoidLambda` rule to convert it to a lambda. Note that this also creates the corresponding interface.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        {
            a = x; g();
        }
        a = y;
    }
    void g() { a = b = 0; }
}
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() { public void apply() {
            a = x; g();
        } }.apply();
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(); }
```

In the next step, the input parameters of the generated lambda are extracted by refactoring `extractInVariables`. `x` is one such parameter, as it is a local variable declared outside of the lambda, but field `a` can be accessed inside the class even with the current unqualified reference.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() { void apply() {
            a = x; g();
        } }.apply();
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(); }
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() { void apply(int x) {
            a = x; g();
        } }.apply(x);
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(int x); }
```

The last step in `extract segment` is to convert the lambda to a method with the `extract` rule. Although removing the interface that is no longer used is formally not a part of this rule, such a refactoring could be easily defined, thus we omit it from the example code to improve readability.

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        new F() {
            public void apply(int x) {
                a = x; g();
            }
        }.apply(x);
        a = y;
    }
    void g() { a = b = 0; }
}
interface F { void apply(int x); }
```

```
class A {}
class B extends A {
    int a, b;
    void f() {
        int y; int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

Since the first part of `lift`, which is responsible for segment extraction, has been completed, we now proceed to the second subtransformation. This is the – also composite – `cascadedLift` that lifts the method created in the previous phase to the superclass. First, field `a` used in `h` is lifted using the `lift` rule.

```
class A {}
class B extends A {
    int a;
    int b;
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a;
}
class B extends A {
    int b;
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

In the next step, method `g` – referenced in `h` – is targeted, on which we recursively call the composite `cascadedLift` refactoring. Its first step is to select field `b` used in `g` and lift it with the `lift` rule.

```
class A {
    int a;
}
class B extends A {
    int b;
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a;
    int b;
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

Since `g` has no more dependencies from its enclosing class, we can lift it to the superclass with the `lift` rule.

```
class A {
    int a, b;
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a, b;
    void g() { a = b = 0; }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void h(int x) { a = x; g(); }
}
```

With the previous step, we completed the transformations needed to lift the dependencies of `h`, so now we can lift `h` itself using the `lift` rule. Thus we performed the second, method-lifting part of our initial refactoring. Since this was the last one, in this step the whole transformation terminates successfully.

```
class A {
    int a, b;
    void g() { a = b = 0; }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
    void h(int x) { a = x; g(); }
}
```

```
class A {
    int a, b;
    void g() { a = b = 0; }
    void h(int x) { a = x; g(); }
}
class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
}
```

6 Future Work

In previous sections we presented a general outline for the adaptation process. Although we aimed to make the case study as constructive as possible, comprehensive support for the object-oriented paradigm is yet to be realized. Here we share two natural continuations of our research which may improve upon this aspect.

6.1 More Schemes and Case Studies

The main concept behind the discussed framework is the notion of refactoring schemes. Therefore it would be beneficial to examine the scheme construction method in finer detail. For example, it would be interesting to conduct more case studies and to compare the different schemes obtained from them. Moreover, the relationship between the top-down and bottom-up approaches also raise additional questions. For example, could results from the two be unified?

6.2 Verification

After an established set of refactoring schemes is constructed, research could proceed with formal verification. Considering the current verification backend, this would mean that almost all language artifacts, including schemes, levels of equivalence and metatheory would need to be formalized in a model compatible with the chosen operational semantics of the target language. Ideally, schemes could be verified manually by structural induction, while scheme instantiation, that is, conformity to scheme contracts would become automatically verifiable.

7 Conclusion

In this paper we presented a proposal for adapting a domain-specific refactoring language from the functional to the object-oriented programming paradigm, using Java instead of Erlang as a representative.

As part of this task, we briefly introduced the original refactoring framework and discussed its description language as well as its verification technique. We also gave an overview of related research.

Then we approached the problem from a high-level perspective, presenting our reasoning about how the adaptation process shall be carried out. We showed how and why the choice of target language and paradigm arose, then discussed how a multilayered definition of equivalence, or even a partial ordering can help to characterize the behavior-preserving property of refactorings in a more intuitive way. We also presented two iterative methods for synthesizing new transformation schemes in the form of the top-down and bottom-up approaches.

Using the latter, we conducted a complex case study where we showed the decomposition of a compound refactoring rule called *lift segment*. With the goal of reconstructing this transformation inside the adapted framework, we began to discuss how different parts of the system should be extended to achieve this target.

In this process, we added new elements to the description language, identified suitable semantic functions and predicates for the target language metatheory (including the notion of inter- and intrahierarchy-reachability) and proposed a set of generalized refactoring schemes. To conclude the case study, we presented formal, scheme-based definitions for decomposed building blocks of the original refactoring, and demonstrated them by the stepwise transformation of a concrete program. Finally, we listed future research directions.

Based on the case study, we conclude that the first steps towards adapting the scheme-based refactoring approach to OOP have been successful: we were able to express a complex Java refactoring in the modified language. As part of this, we found a suitable decomposition for this transformation, and then we were able to generalize schemes from the resulting microsteps. By constructing an appropriate program equivalence, a description language and a metatheory, we managed to make the identified schemes definable. We have seen that these schemes are already suitable for expressing the initial base refactoring. Their generality obviously still falls short, but we hope that a more comprehensive scheme library can be built with the presented technique in the future.

References

[1] Aiken, Alexander. Cool: A Portable Project for Teaching Compiler Construction. *SIGPLAN Not.*, 31(7):19–24, July 1996. DOI: 10.1145/381841.381847.

[2] Bogdanas, Denis and Roşu, Grigore. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, New York, NY, USA, 2015. ACM. DOI: 10.1145/2676726.2676982.

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

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

[5] Corliss, Marc L., Furcy, David, Davis, Joshua, and Pietraszek, Lori. Bantam Java Compiler Project: Experiences and Extensions. *J. Comput. Sci. Coll.*, 25(6):159–166, June 2010. URI: http://dl.acm.org/citation.cfm?id=1791129.1791160.

[6] Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN: 0201485672.

[7] Garrido, Alejandra and Meseguer, Jose. Formal Specification and Verification of Java Refactorings. *Proceedings - Sixth IEEE International Workshop on*

Source Code Analysis and Manipulation, SCAM 2006, pages 165–174, 2006. DOI: 10.1109/SCAM.2006.16.

[8] 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 of the 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.

[9] Horpácsi, Dániel, Kőszegi, Judit, and Thompson, Simon. Towards Trustworthy Refactoring in Erlang. 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 83–103. Open Publishing Association, 2016. DOI: 10.4204/EPTCS.216.5.

[10] Kalleberg, Karl Trygve. *Abstractions for Language-Independent Program Transformations.* PhD thesis, University of Bergen, Bergen, Norway, 2007. URI: http://hdl.handle.net/1956/3287.

[11] Leitão, António. A Formal Pattern Language for Refactoring of Lisp Programs. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 186–192, 2002. DOI: 10.1109/CSMR.2002.995803.

[12] Li, Huiqing and Thompson, Simon. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Fundamental Approaches to Software Engineering*, pages 501–515, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-28872-2_34.

[13] 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.

[14] Roşu, Grigore and Ştefănescu, Andrei. From Hoare Logic to Matching Logic Reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, Volume 7436 of *LNCS*, pages 387–402. Springer, Aug 2012. DOI: 10.1007/978-3-642-32759-9_32.

[15] Schäfer, Max, Verbaere, Mathieu, Ekman, Torbjörn, and de Moor, Oege. Stepping Stones over the Refactoring Rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag. DOI: 10.1007/978-3-642-03013-0_17.

[16] Schäfer, Max and de Moor, Oege. Specifying and Implementing Refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 286–301, New York, NY, USA, 2010. Association for Computing Machinery. DOI: 10.1145/1869459.1869485.

[17] Sommerville, Ian. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010. ISBN: 0137035152.

[18] Stefănescu, Andrei, Park, Daejun, Yuwen, Shijiao, Li, Yilong, and Roşu, Grigore. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 74–91, New York, NY, USA, 2016. ACM. DOI: 10.1145/2983990.2984027.

[19] Verbaere, Mathieu, Ettinger, Ran, and Moor, Oege. JunGL: a Scripting Language for Refactoring. In *Proceedings – International Conference on Software Engineering*, Volume 2006, pages 172–181, 01 2006. DOI: 10.1145/1134311.

[20] Visser, Eelco and Benaisse, Zine-el-Abidine. A Core Language for Rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422–441, 1998. DOI: 10.1016/s1571-0661(05)80027-1.