Verification and Application of Program Transformations

Dániel Horpácsi

Supervisor: Zoltán Horváth

Eötvös Loránd University, Doctoral School of Informatics
Foundations and Methodologies of Informatics

Head of School: Erzsébet Csuhaj-Varjú, DSc.
Head of Program: Zoltán Horváth, PhD.

This dissertation is submitted for the degree of
Doctor of Philosophy

August 2018

To my beautiful daughter, Eszter.

Acknowledgements

First and foremost, I thank my wife and my family for their unconditional love and support all the way. Secondly, I thank my supervisor, Zoltán Horváth, who was always keen to help me with all my professional or personal dilemmas, even in his busiest times. Special thanks to Simon Thompson, my advisor during the two semesters I spent in Canterbury. Perhaps not purposely, Simon taught me a lot about life and science. I will always think of my Canterbury visits with gratitude and love in my heart. I thank the entire Erlang team at ELTE, particularly Melinda Tóth, István Bozó and Róbert Kitlei, for all those refreshing discussions and fruitful cooperation. Last but not least, I am grateful to Tamás Kozsik and Máté Tejfel, who took the role of being my advisor when I was in need of wise words.

Contents

1 Introduction

"A programming language is low level when its programs require attention to the irrelevant." (Alan J. Perlis)

As a bachelor student, I learned from my future doctoral supervisor that one of the main responsibilities of computer scientists is to make appropriate abstractions. New abstractions induce new concepts, and expressing our ideas with those is expected to be more comprehensible or more efficient in some way. Essentially, the introduced abstractions shall improve the expression of thoughts, algorithms, or ideas, and should positively affect how we use them. This fundamental idea determined the way I understood problem solving ever since.

My doctoral results are dealing with solving complex problems by carefully identifying and precisely defining problem-specific abstractions. This involves understanding and establishing syntactic and semantic relationship between the original problem and the solution. The first two theses literally introduce new languages (with new abstractions) that bring improvements on how effectively and reliable we can express solutions to complex problems. The third thesis is also language-oriented, yet it is more about identifying and exploiting capabilities of well-known language processors.

From the value proposition point of view, my doctoral work improves on the reliability, the trustworthiness of refactoring transformation definitions, and in addition, it showcases a special application of the capabilities of a refactoring system by using it for implementing language extensions. The approach I use along with the voice of the presentation is highly determined by the fact that my entire doctoral research was focussed around static analysis and transformation of software source code, and the fact that I taught Programming languages, Compilers, Formal semantics and Formal verification throughout the past years.

The principal case study language for my methods is Erlang, a functional programming language. Erlang enjoyed a special interest around 2007, the year I joined the refactoring-focused university research project supported by Ericsson Hungary; many of my results have been obtained in the scope of this project.

Besides working with static analysis and transformation in functional languages, I also participated in research of Software Defined Networking (SDN). As a matter of fact, I wrote the first prototype of T4P4S [40] (Translator for P4 Switches), a retargetable compiler for the P4 language. This work of mine is not closely related to this dissertation.

Contributions

I started my research career as a member of the Erlang refactoring project at ELTE. I contributed to the design of the analysis and transformation framework [35, 6] of our Erlang refactoring tool, as well as to the design and implementation of various concrete analysis [32, 55, 7] and transformation [45] steps. I also took part in a project that dealt with automatic parallelisation of Erlang programs [5, 4], this latter guided my attention towards high-level refactoring definitions. This work and these results provided me a solid background knowledge and long-lasting interest to accomplish my doctoral results.

This dissertation presents three theses in three main chapters.

The first two theses deal with dynamic and static verification of refactoring transformations. The dynamic verification approach (Chapter 3) is based on property-based testing of refactoring with randomly generated programs. My main contribution here is the method of synthesising data generators for L-attribute grammars [15], and the attribute grammar I composed for a well-formed subset of Erlang. I also contributed to the application of the generator for property-based testing of analysis and transformation components [67]. A few years later I shifted my attention towards static, formal verification of refactoring transformations (Chapter 4). With the help of Judit Kőszegi, I designed a refactoring specification language [29] that allows for defining executable transformations that are automatically verifiable for semantics-preservation. For the demonstration of the applicability of the approach, we composed a complex case study refactoring [30], the verification thereof will be discussed in Judit's doctoral dissertation.

While the first two chapters report on results related to verification of refactoring, the third chapter presents an unusual application of refactoring frameworks. Chapter 5 uses slightly customised analyses and program transformations to implement new language features with a refactoring system. The main contribution is a detailed explanation of how a refactoring framework is used for extending languages, and some complex case studies of features implemented with this method [28]. I also contributed to the exploitation of my language extension method for embedding workflow description operators in Erlang [39]. Although the language extension method is based on refactoring transformations, those define translational semantics to the new language constructs, therefore they are not subject to verification techniques discussed in the first two theses.

In the following, Chapter 2 provides a brief overview of the analysis and transformation framework I worked with, both regarding verification and implementation. Then, the main chapters detail the main results of the doctoral research. Finally, Chapter 6 will summarise and conclude.

2
Framework for analysis and transformation

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." (Martin Fowler)

In this chapter we briefly overview some background to the presented results. We first introduce the programming language used as a case study language for the methods presented in this document, then we describe the analysis and transformation system built for this language. In addition, we identify some key features of the system the thesis results build on, with focus on semantic analysis and transformation libraries.

Erlang

Most results presented in this document are somehow connected to the functional programming paradigm and to the Erlang [11] programming language in particular. Static analysis and transformation algorithms, domain specific language implementations, almost all prototypes of my methods have been implemented in this programming language. The most obvious reason for this is the analysis and transformation system available for this language, which I contributed to during my research. Note, however, that this does not mean at all that my methods are completely Erlang-specific; rather, they may be partly restricted to languages showing similar features.

Erlang is a concurrent, impure, functional programming language. It has been designed for creating fault-tolerant, scalable telecommunication systems, and is being used by the software industry to develop robust server applications. Programs written in Erlang are composed of files, which consist of a set of forms encapsulating series of expressions. Files define modules, and forms define program entities such as functions and records.

Erlang is eagerly evaluated, and it is strongly but dynamically typed. Because of the dynamic nature of the language, it is rather challenging to provide static analysis and correct refactoring for programs written in it. One possible approach to Erlang static analysis is proposed by RefactorErl [70, 69, 6]. The refactoring system we statically and dynamically verify in the following sections has been written for Erlang, in Erlang.

2.1 Analysis

The verification and transformation methods discussed in the document have all been implemented for, or on top of, the above-mentioned static source code analysis and transformation tool, called RefactorErl. This refactoring system implements a complex software architecture for statically analysing, representing, inspecting and transforming Erlang programs.

Being a complex language processor, this refactoring system implements thorough lexical, syntactic and static semantic analyses of source code, and turns the obtained information into a three-layered graph representation stored in a database. Not only these solutions are theoretically founded, but also, the implementation is capable of handling industrial-scale source code, and it has already been applied to analyse source code of real-world telecommunication switch software. Beside data-flow, control-flow and binding analysis, the tool is able to perform dependency analysis, duplicated code detection and code clustering, to mention but a few of the capabilities.

Syntactic analysis. The refactoring tool uses its own definition of the Erlang language on all levels, from syntax to semantics. This definition was designed to be extensible, and easily adaptable to newer versions of the Erlang language [36]. The lexical and syntactic analysers are automatically generated from a special, lightweight description, which defines both the concrete and the abstract syntax of the language. Based on this description, we can synthesise a simple lexical analyser and a yecc grammar description, and in addition, a dedicated module supporting syntax subtree construction [34].

Semantic analysis. Static semantic analysis is implemented in a novel way in this tool. Syntactic objects added to the program model are automatically analysed for static semantic properties. Binding, data-flow [71] and control-flow analyses are implemented as standalone analyser modules capable of running in parallel, and the synthesised information is added to the program model in terms of semantic objects and relations.

Semantic analysis is incremental. That is, in case if the model needs to be updated due to a change in the syntactic part, only those fragments are analysed for semantics that have been modified. Semantic analyser modules are able to handle deletion and addition of syntactic objects, and they adjust the semantic layer of the model incrementally, without performing any unnecessary computation. On the other hand, there are analysis steps that cannot be done incrementally for a reasonable effort. Message-passing analysis and dynamic function call analysis [32] are two examples that, in case of changes, need to be rerun from scratch on the whole database.

2.2 Representation

The way programs are represented in the tool may have a high influence on complexity and efficiency of analysis and transformation algorithms. The role of semantic program models is to capture the syntax and static semantics of the code in a compact yet efficient way. This is achieved via static semantic analysis [72] that enriches the abstract syntax tree with semantic nodes and edges representing context-sensitive properties. A good program representation simplifies code understanding, further static analysis and program transformation.

The syntax tree The fundamental part of any program model is the syntax tree. This is obtained by performing syntactic analysis (parsing) on the lexically analysed and preprocessed token stream. We need to make a distinction between concrete and abstract syntax. Concrete syntax is the language that the compiler accepts and the sentence of which the programmer writes down. This is the language we write a parser for, which results the so-called concrete syntax tree (CST). On the other hand, language processors (compilers and refactoring systems) usually operate on the abstract syntax, which is a distilled version of the language: symbols required solely for parsing and readability are left out, only those syntax elements are kept that influence program semantics.

Both in compilers and refactoring tools, the concrete syntax tree (CST) is implicitly built by the parsing process (performing reductions according to the production rules of the concrete syntax grammar), while the abstract syntax tree (AST) is explicitly constructed by routines associated with the concrete prodution rules. We discuss concrete and abstract syntax in more detail in Section 3.3.

The semantic program graph (SPG) Static semantic analysis uncovers context-sensitive properties of programs which cannot be represented in ordinary syntax trees. More expressive notations like annotated abstract syntax trees (AAST) and higher-order abstract syntax trees (HOAST) can accommodate extra information, but in RefactorErl, we decided to build a semantic graph model.

The semantic program graph (SPG) is an extension of the abstract syntax tree; essentially, it is a rooted graph with labelled, indexed and directed edges. On top of the AST, it includes semantic nodes and semantic edges. We sometimes call nodes objects, emphasising that all nodes have a unique id and an attribute set. In analysis and transformation algorithms, nodes are referred to by their unique id, and most of our algorithms are impure: they suppose a globally accessible graph object, parts of which can be manipulated via node references.

The methods we present in the forthcoming chapters assume the SPG as the underlying program model for analysis and transformation. In particular, we assume the model be represented as a mutable tree or graph, with references to every syntactic and semantic node (object).

Worth mentioning that, unlike in compilers, lexical nodes in the refactoring system store not only the texts of the code elements, but also the comments and the white-spaces around them. This allows the tool to restore the code according to its original appearance following any transformation steps.

Example 2.2.1. The following example demonstrates a simple Erlang program and its corresponding Semantic Program Graph (syntactic and semantic layers).

```
-module(example).
sum([]) -> 0;
sum([H|T]) -> Sum = sum(T), H + Sum.
```

Figure 2.1 shows the corresponding program graph. The semantic layer is on top of the syntactic layer: modules, functions and variables (name bindings in different name spaces) are treated as semantic entities and get a dedicated semantic node created in the graph, whilst the semantic nodes are connected to the syntactic nodes with labelled edges. This kind of representation makes it a one-step traversal from a semantic entity to localise its definition and its references, which simplifies queries and transformations.

2.3 Transformation

Transformations in RefactorErl are implemented as syntax tree manipulations written in Erlang. Although these algorithms build on the graph query [47] and tree construction libraries [35] created for the semantic program graph, they are pretty low-level; Chapter 4 addresses this issue.

Refactoring transformations are implemented as consecutive actions of graph queries and tree rewritings. Essentially, before any modifications, the side-conditions of refactoring are checked; these guarantee that the transformation preserves the behaviour and the semantics of the refactored program. If the conditions are met the system starts manipulating the syntax tree in multiple syntactic transactions, each followed by semantic analysis. Following the transformation, the framework pretty-prints the modified model into its textual form (source code) and saves it in the file system.

Graph queries. Node and property lookup is realised by (semantic) graph queries. Node annotations and edge labels are both exploited in such queries, i.e., while finding nodes or properties — paths in the labelled graph are basically described by means of a series of edge labels. The query library builds on top of these traversal paths, and uses language-level concepts for information lookup. For example, one can make queries like "please find me all 2-parameter functions that are in module X, refer to some record R or to some other function Y, and do not have side-effects", but apparently, the syntax is not English-like, it is given by calls to functions in the query library.

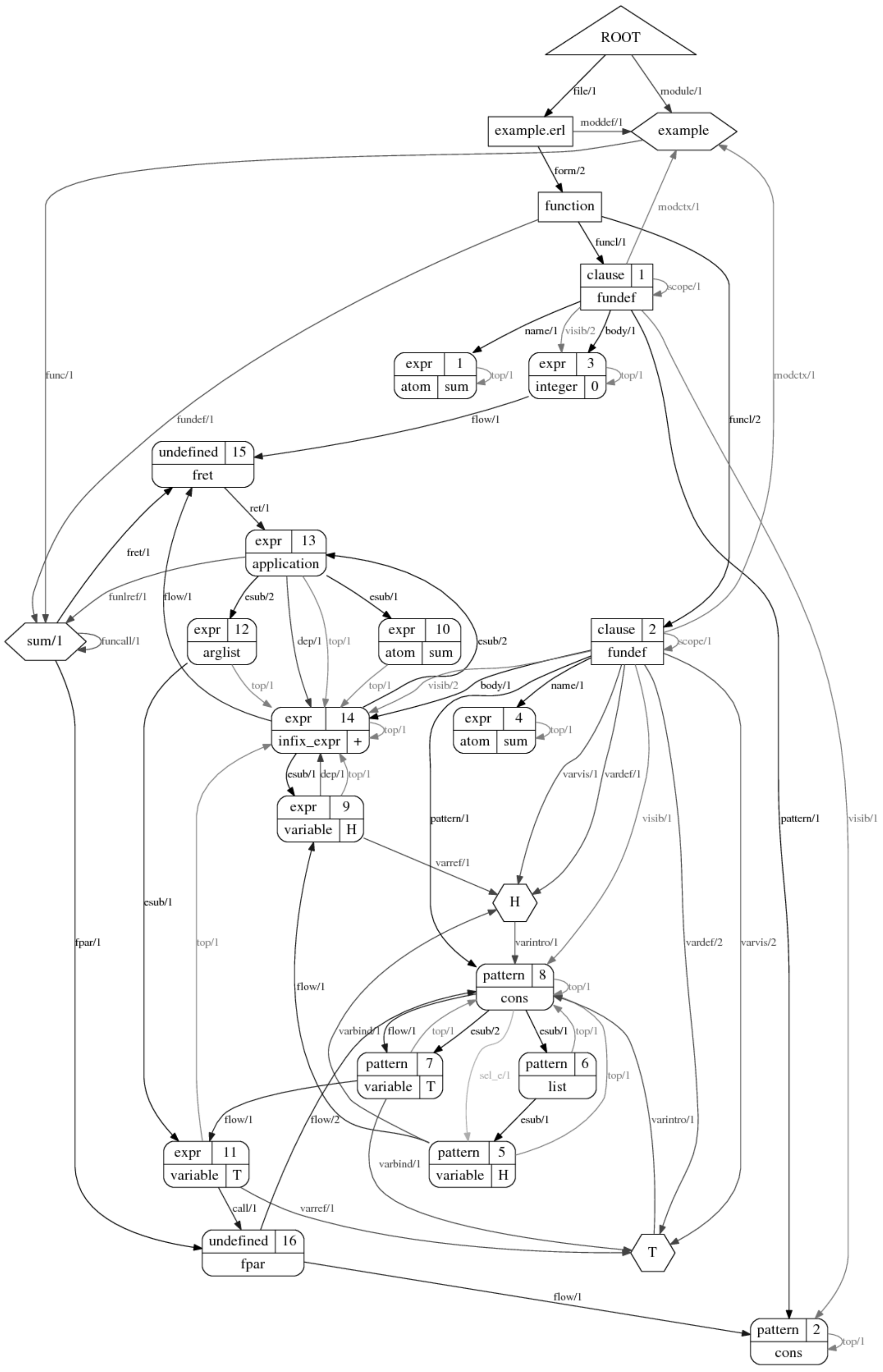


Figure 2.1: A semantic program graph

Tree construction. RefactorErl, in order to make tree operations completely independent of the actual low-level representation of the graph, provides a library that hides implementation details behind a simple interface. Via this interface, one can intuitively add or delete graph nodes, get or update node annotations, and add or remove directed, labelled edges between two nodes.

Creating subtrees, and making replacement by detaching and attaching nodes is cumbersome in the low level, but the framework lets us construct syntax trees based on abstract syntax. In this library, one can reuse existing nodes to build more complex out of them. That is, in the description of the desired tree, nodes can be specified by their original identifier, or with a simple Erlang term representing an abstract syntactic construct composed of other nodes. This set of construction functions, similarly to the semantic analysis and pretty-printing, is based on a predefined abstract syntax of the language.

We emphasise that thanks to the automatic and incremental static semantic analysis in RefactorErl, syntax manipulation is always followed by the automatic re-adjustment of the semantic layer, without manual graph extensions.

2.4 Correctness

Refactoring, by definition, is semantic-preserving program transformation. However, in general programmers associate refactoring with program changes that improve appearance or performance of their software. In this document, we do not consider the usefulness of refactoring transformations: the focus is on whether the transformation is guaranteed to preserve program behaviour (or semantics), whilst non-functional properties are neglected.

A refactoring is said to be correct if it implements a program transformation that is definitely semantics-preserving. In widely used refactoring systems, transformations implementations are not formally verified, and there is always concern about whether they are correct. A refactoring failing to preserve the meaning of the code may cause significant expenses in a project; in some companies, programmers are discouraged to do refactoring at all. The concern is well-founded: a study [14] comparing the Java refactoring steps in Eclipse and NetBeans found more than 20 errors in the implementation of common refactorings in each tool.

There are several static and dynamic approaches to refactoring verification. Refactoring can be tested with test suites or on randomly generated input programs, although the latter requires a method for checking behaviour-preservation. In Chapter 3 we present a solution that is based on checking program equivalence with randomly generated input values. Although this kind of verification may bring confidence in correctness to some extent, the randomised nature makes it fragile from the coverage point of view. Let us be honest, it is far from making the refactoring trustworthy for the advanced user.

Dynamic verification can be improved when combined with static proofs. We can incorporate formal semantics of the language in verifying concrete (possibly randomly generated) applications of the refactoring for semantic-preservation. This may not test, but prove that one particular application of the refactoring (e.g. on program P) was definitely correct (resulting in a program P' semantically equivalent to P), but it is left open whether it would be correct on another P. Apparently, we cannot enumerate all the possible programs.

$$\begin{tikzcd}\boxed{\text{P}} \arrow[r, "R"] & \boxed{\text{P'}}\end{tikzcd}$$

The ultimate correctness verification is proving that the refactoring R is correct for any possible P, such that applying R on any program P refactoring R preserves its semantics. This can be achieved by static, formal verification of refactoring definitions, which requires the refactoring be defined in an abstraction level and formalism that is amenable to formal verification, as well as the programming language semantics and the semantic equivalence be precisely defined in a formal notation. This approach is addressed in Chapter 4.

3 Property based testing of refactoring systems

"Discovering the unexpected is more important than confirming the known."
(George E. P. Box)

This chapter investigates dynamic verification of refactoring correctness. Although testing, unlike formal proofs, cannot ensure the absence of errors, it is seen as the most important and widespread verification technique for large-scale software. Via testing, we can show that the program's run-time behaviour is correct in a significantly large number of cases. The more test cases we check, the more confidence testing brings, or in other words, the more chance we have to uncover incorrect behaviour. Indeed, if the test suite includes some obscure corner cases as well, testing may be just enough to provide the required confidence in the correctness.

In its simplest form, testing is based on the inspection of output values (and side effects) on given input values. We naturally associate programs with mathematical functions (or relations) on special domains (this may be seen as a denotational semantics of the program), and understand testing as checking of some particular elements in this function. There is an obvious limitation: the input domain of the computation is typically infinite, which makes it practically impossible to enumerate all the possible input values and perform exhaustive testing to prove the program error-free.

In case-based testing, a number of test cases are composed (either via white-box or black-box, i.e. based on the program or based on its specification), which are expected to cover all the important (and representative) input-output pairs, and execute as many paths in the program as possible. The result of this process is a so-called test suite, which can be used for unit testing and regression testing, and can be regarded as a partial input-output specification of the program. In the case of refactoring transformations, such a test collection would contain a number of source code pairs showing the result of a particular transformation on a piece of code. There is an apparent connection between the original and the transformed program: they have to be semantically equivalent.

Property-based testing (PBT) is a generalisation of the traditional case-based testing: rather than enumerating a huge number of input-output pairs, we specify the expected behaviour of the program in a precondition-postcondition fashion. Namely, concrete input values are generalized to pre-conditions on the input domain (data set), and concrete output values are replaced by post-conditions on the resulting data.

For very simple, total functions, the input domain can be synthesised based on the type of the function, but in the rest of the cases it has to defined separately with so-called data generators. The system performs the testing by randomly generating the elements of the input domain by consecutive calls to the input data generator. Note that when testing complex programs operating on complex data, both specifying the data generator and establishing the formal property may be pretty challenging.

Problem statement

In refactoring systems, both the input and the output is program source code. When specifying the correctness of a refactoring transformation, the precondition would ensure that the input is a compilable source code, whilst the postcondition tells if the resulting source code is semantically equivalent to the original one. If we can make sure that the data generator for the input domain only generates compilable programs, no further precondition is necessary. Now the question is how to create a PBT data generator for well-formed Erlang programs and how do we check program equivalence. Let us first focus on the former one.

Suppose that *P* is the set of syntactically-valid Erlang programs, which can be processed by a refactoring system. When specifying the correctness property of a refactoring ($\mathit{refactor} : \mathit{Program} \rightarrow \mathit{Program}$), we restrict the input domain to semantically valid (well-formed) programs, and only require the transformation to preserve the semantics of such programs. Note that the refactoring may or may not transform an ill-formed (e.g. not compilable) program properly. The following formula defines the above property:

$$\forall p \in \mathit{WellFormedProgram} : \mathit{equivalent}(p, \mathit{refactor}(p))$$

The same property can easily be expressed in QuickCheck, a property specification language, resulting in the following expression:

```
?FORALL(P, well_formed_program(), equivalent(P, refactor(P))
```

This reads "for all well-formed programs, the program and its refactored version are semantically equivalent"[1], which corresponds to the formal statement expressed in first-order logic. The property seems to be straightforward, but relies on two complex concepts: the language of well-formed programs and a predicate that tells whether two programs are semantically equivalent. How do we define these? This is the problem this chapter investigates.

[1]The refactored (transformed) variant may be identical to the original program if the transformation's precondition is not fulfilled, but this is accepted since two identical programs are semantically equivalent.

Structure of the chapter

This chapter is structured as follows. First, Section 3.1 overviews related work in the field, including both grammar-based data generation and automatic testing of refactoring. Then, in Section 3.2, the underlying theory of testing properties and generators is discussed, which clarifies the abstraction layer we can build on. Section 3.3 discusses the method we developed for attribute grammar-based test generation in the QuickCheck property-based testing framework. As a case study, Section 3.4 demonstrates the method by defining a subset of Erlang, and Section 3.5 discusses how the grammar-based generator has been used in testing of refactoring engines. Section 3.6 identifies some limitations and future work, and sums up the results of the chapter.

3.1 Related work

Both grammar-based test generation and automatic testing of refactoring transformations have been of interest of both academia and industry for a long time now. Automatic test case generation for dynamic verification is a big challenge in most programming paradigms, and if, in addition, the tested program consumes programs, the problem becomes even more challenging. In this section, we briefly overview the related work in grammar-based data generation and automatic testing of refactoring engines.

Grammar-based test generation. In software technology context-free and attribute grammars are mostly used to recognise sentences of languages, but it is not a novel idea to employ grammars to generate sentences of languages. Enhanced context-free grammars have been identified long ago as concise and powerful formalism for defining various kinds of complex and structured data. With the ability of automatic sampling of generated languages, they can outstandingly support testing of systems defined on structured input domains.

Since its introduction in the early 1970s, grammar-based test generation has become well-established: proven on industrial projects and widely published in academic venues. It has been used in dynamic verification of various hardware and software systems [16, 48], including network processors and integrated circuits. Not surprisingly, given that programming languages are typically formally defined with grammars, the approach has also been applied to generate random programs for testing meta-programs, such as parsers, compilers and refactoring engines [61, 49].

In the related work, which intensified in the years following the millennium, various domain specific languages and implementations have been created for context-free grammars and variants of attribute grammars. Some of these have been designed to primarily support extensible programming with special variants of attribute grammars [77, 18], while others strictly aim at providing assistance for test data generation [27, 24]. Different applications demanded for different extensions to the classical grammar formalisms.

The approaches mainly differ in what additional features to attribute grammars they support (e.g. reference attributes, higher-order attributes, circularity, probabilities), whether they offer standalone or embedded implementations, in which programming paradigm they interpret the grammars, and in what terms the grammar is given a semantics. From the semantics perspective, attribute evaluation is a central question, since uncovering hidden dependencies among attributes may be very difficult, while from a more syntactic point of view, modularity and compositionality (supporting feature-oriented language development) are the primary goals of developing attribute grammar languages. Lämmel [38] made an extensive survey on grammar-based software, which identifies grammarware an important principle in software engineering. Indeed, attribute grammars have important role in language specification, language design, implementation, analysis, and in language sampling for different purposes in general.

Attribute grammars in Erlang. In the functional paradigm, and particularly in Erlang, there are at least two developments definitely worth mentioning. The very first grammarware solution for Erlang QuickCheck was the component called EQC Grammar, a compiler that could transform a context-free grammar into a QuickCheck data generator. Although this proved to be a useful tool for generating some simple kinds of data (such as context-free protocol messages), it was not expressive enough to generate data with context-sensitive properties.

A few years later, the QuickCheck development team created support for attribute grammars [52]. The expressiveness of this solution is pretty similar to mine (in fact, they can express a larger set of grammars), but the way we formalise and interpret grammars is very different. In their solution, grammars are embedded into Erlang, not compiled as a standalone language; the context-free syntax is defined in terms of type specifications, whilst attribute computations are written as functions in a special syntax. Conditions on attributes are handled in a very unique way, but they implement implicit inheritance similar to my solution; they call it chained attributes. The formalism is more modular then mine, but it is pretty hard to embed a formal attribute grammar into their language. In contrast, my generator language is basically a textual representation of a formal L-attribute grammar, meaning it is straightforward to script an already composed grammar in my language.

Testing of refactoring. Large-scale refactoring systems are usually tested with an extensive set of manually written test cases. Fortunately, a lot of effort has been put into property-based testing of refactoring engines, which can significantly increase coverage. Beside providing meaningful input programs for the refactoring, the main problem of automatic refactoring verification is checking whether the output is correct w.r.t. the input program. In general, it is not decidable if two programs are semantically equivalent, therefore, other, static semantic properties are checked for equivalence or desired change.

Important results were reported in [14], presenting a QuickCheck-inspired, custom-designed generator library in which a Java program generator is instantiated. They check different test oracles on refactorings executed on random programs, similarly to our approach. For Erlang refactoring, Thompson and Li[42] made the first important steps towards automatic testing of transformations. Although they used randomly generated transformation commands, the input code was pre-written, not generated. They checked correctness via checking binding structures and other lightweight properties of programs.

3.2 Property-based testing with QuickCheck

Before diving deep into the definition of the refactoring correctness property and the corresponding data generator, let us overview what we cook with. In Erlang, just like in a number of other programming languages, property-based testing is easiest done with QuickCheck. QuickCheck is a language and toolset for stating and checking properties of programs. The implementation uses the host language's variables and boolean expressions for composing first-order logic formulas for the properties.

3.2.1 Properties

Properties are many-sorted first-order logic statements that capture aspects of partial correctness of functions. Variables in these formulas can take values of (sub-types of) data types definable in the language. That is, variables' sorts are restricted by the programming language's type system.

Example 3.2.1. Consider the example mentioned in the problem statement. Expressed in first-order logic, the domain of discourse may be either syntactically valid programs or semantically valid (well-formed) programs. When opting for the former, the property would not necessarily hold, because the behaviour of refactorings on semantically ill-formed programs is not defined. Nevertheless, if we restrict the universe to well-formed programs, the property would hold.

$$\forall p(\textit{equivalent}(p, \textit{refactor}(p)))$$

When choosing syntactically valid programs as universe, with implication in first-order logic, we can explicitly restrict the scope of the property, making it hold.

$$\forall p(\textit{wellformed}(p) \rightarrow \textit{equivalent}(p, \textit{refactor}(p)))$$

As mentioned already, QuickCheck uses the typed (many-sorted) variant of first-order logic, variables' sorts are explicitly stated. Let us rephrase the formulas typed, to make universes explicit.

$$\forall p \in \mathit{Program} : \mathit{wellformed}(p) \rightarrow \mathit{equivalent}(p, \mathit{refactor}(p))$$

Or equally: $$\forall p \in \lceil \mathit{wellformed} \rceil : \mathit{equivalent}(p, \mathit{refactor}(p))$$

This formula expresses the property the way we want it. In first-order logic, there is no difference between restricting the universe and introducing a strong precondition, but in QuickCheck, there are some technical considerations, and in general, we have to opt for the second option, where the domain is smaller. The reason is that there is no connection between the implications and the data generators, random values are generated independently of preconditions. Data dissatisfying the preconditions is simply excluded from the actual test set. For strong preconditions on complex domains, QuickCheck would randomly generate extensive amounts of wrong (w.r.t. the precondition) test cases, without doing actual verification of the property. Therefore, in QuickCheck we shall use well-specified generators with weak preconditions, rather than general generators with strong preconditions.

Example 3.2.2 (Advanced generators instead of strong preconditions)**.** Consider the following generator that produces random even numbers.

```
?SUCHTHAT(X, int(), X rem 2 == 0)
```

It seems obvious to ask the testing framework to generate random integers and filter out the odd ones, but beware that the system cannot magically produce the values we want. What will happen is that it will keep trying random values one after the other until it finds one that passes the precondition. If a large percentage of all the possible values do not pass the test, then a better strategy is to write a more advanced generator that only generates 'good' cases:

```
?LET(X, int(), 2 * X)
```

Properties expressed in QuickCheck are explicitly typed formulas, all variables are associated with a corresponding domain. These domains are determined by QuickCheck data generators. In Erlang QuickCheck, implemented with macros and functions, we would write the following expression to express the refactoring correctness property:

```
?FORALL(P, ?SUCHTHAT(P0, program(), wellformed(P0)),
        equivalent(P, refactor(P)))
```

Or rather than building the implication into the generator, we can write it explicitly in the property:

```
?FORALL(P, program(),
        ?IMPLIES(wellformed(P), equivalent(P, refactor(P))))
```

However, as we just emphasised it, it is more practical to opt for a more refined domain instead of the strong precondition:

```
?FORALL(P, wellformed_program(), equivalent(P, refactor(P)))
```

With this, we ended up with the property we sketched in the problem statement. Now let us overview the set of properties and generators available in QuickCheck.

Basic properties. After the introductory example, we overview all abstractions available for property construction. We define what we mean by a property in QuickCheck, then we show the connection between first-order logic specifications and QuickCheck properties.

Definition 3.2.1 (QuickCheck property)**.** Suppose that *BoolExp* is the set of boolean expressions in the programming language. Then, the set of basic QuickCheck properties (*Prop*) is defined as follows:

- $\mathit{BoolExp} \subseteq \mathit{Prop}$
- If $B \in \mathit{BoolExp}$ and $P \in \mathit{Prop}$, then $(B \rightarrow P) \in \mathit{Prop}$
- If G is a generator for type a and p is a function of type $a \rightarrow \mathit{Prop}$, then $(\forall G : p) \in \mathit{Prop}$
- If G is a generator for type a and p is a function of type $a \rightarrow \mathit{Prop}$, then $(\exists G : p) \in \mathit{Prop}$

Properties specified as a boolean expression are called primitive properties, while the inductive cases (implies, for all, exists) are called property combinators. The concrete syntax of these combinators depends on the programming language that implements the system. In Erlang, three macros are used to compose such properties: `IMPLIES`, `FORALL` and `EXISTS`.

Proposition 3.2.1. *Basic properties are specifications expressed in many-sorted first-order logic formulas.*

Proof. Boolean expressions in the programming language correspond to simple formulas over many-sorted first-logic without quantifiers. The three main property combinators correspond to the implication operator, the universal and the existential quantifiers in first-order logic. $\square$

Remark. In some implementations of QuickCheck (e.g. in Haskell), there are property combinators for the logic connectives *and* and *or* as well.

Additional property combinators. For technical reasons, there are a few more property combinators in Erlang QuickCheck. These do not affect the logic formula the property checks, but may influence the checking strategy. For example, `FORALL_TARGETED`, a variant of `FORALL`, uses advanced techniques to control test case distribution by optimising on some target metrics, instead of using simple randomised test case generation. Furthermore, to tackle run-time errors and divergence encountered during test, `WHENFAIL`, `TRAPEXIT` and `TIMEOUT` can help the property writer handle exceptional scenarios during check.

3.2.2 Generators

"Good generators are often the difference between finding bugs and not finding them."

Data generators are QuickCheck's tool to determine domains of discourse for quantified variables in properties. The simplest data generators are not more than a mapping to a corresponding data type, while more advanced generators restrict domains or control distribution of elements in random selection.

First-order generators

First-order generators take no arguments, or take arguments of the data types of the programming language that implements QuickCheck. In other words, first-order generators cannot be parametrized by other generators.

A QuickCheck implementation provides data generators for the types of the programming language. In the case of Erlang, there are built-in first-order generators for all ground types, such as *binary, bool, char, int, largeint, nat* and *real*. These generators randomly generate a value out of the data type they belong to (i.e. the generator *int* generates a random integer value).

In addition, there are two first-order generators that take concrete values and lift them to generators:

- *return*: Constructs a constant generator for any value in any type in Erlang.
- *elements*: Takes a value from a list of (possibly heterogeneously typed) elements.

Higher-order generators

Unlike first-order generators, higher-order generators take one or more generators as arguments. They can either restrict or extend the generated domain, as well as they can change the probabilities associated with domain elements. We overview most of the generator combinators available in (Erlang) QuickCheck.

Dependent generation. Although Erlang, unlike Haskell, has no support for monads, the abstract type of data generators does have the usual monadic interface: return and binding (return has been mentioned in the previous section as a first-order generator). The binding operator (used as the `LET` macro in Erlang[2]) supports building generators whose domain depends on the value produced by another generator. This way, we can implement monadic composition of generator functions in Erlang, such that a generator depends on the value of another generator.

Generating structure types. QuickCheck allows for generating compound data types, such as lists and tuples in Erlang. There are higher-order generators for this, called *list* and *tuple*, respectively. When used with first-order generators, they will generate homogeneous structures, whilst parametrised with higher-order generators producing union types, they can generate heterogeneous structures, too.

Domain restriction. There are two ways to restrict the domain of discourse for a quantified variable in QuickCheck: via a boolean condition on the value, or via limiting structural complexity (size).

- *suchthat*: If we need to restrict a domain to a subset of elements satisfying some simple properties, the *SUCHTHAT* generator combinator can be used. It can act as a precondition before passing a generator value to a property.
- *sized*: During the testing process, QuickCheck maintains a complexity measure (so-called size) for all data generators. Properly implemented generators take this value into account, such that the bigger the generator size is, the more values the generator selects from. The *SIZED* generator combinator lets the argument generator rely on the actual value of the generation size. For structured data (e.g. syntax trees), the size may restrict the domain in terms of structural complexity (e.g. number of nodes or tree height).
- *resize*: This generator allows us to modify the size attribute of its argument generator.

Example 3.2.3. To get an idea on how generators are constructed, let us consider a simple example. The following generator produces a list of points with two integers as coordinates.

```
list(?LET({X, Y}, tuple(int(), int()), {point, X, Y}))
```

Another example demonstrates the use of the *SUCHTHAT* combinator. The following line defines a not-too-efficient generator that produces random numbers not being prime:

```
?LET(Y, int(), ?SUCHTHAT(X, int(), Y > 1 and X rem Y == 0))
```

[2]`LET(X, E, E2)` is defined as `bind(E, fun(X) -> E2 end)`

Union domains. It is also possible to extend, or rather, unify (merge) domains. Technically, the generation system will choose non-deterministically among the argument generators, but this non-determinism on the domain level means simply taking the union of the argument domains.

- *oneof*: This combinator takes a list of generators and results in a new generator that produces values of the domains of the argument generators with an even probability.
- *frequency*: This combinator is the weighted variant of *oneof*, where each and every argument generator is coupled with a probability. With this, the probability of the elements in the unified domain can be controlled per component domain.

Example 3.2.4. The following generator produces a numbers such that the probability of generating an even number is double the probability of getting an odd one.

```
frequency([ {2, ?LET(X,int(),X*2)}, {1, ?LET(X,int(),X*2+1)} ])
```

We will use domain unification extensively when producing different kinds of syntactic program elements of the same syntactic category. Quite naturally, the generator for the language of expressions is defined in terms of the union of generators for its sub-languages.

Remark (Analysing distribution.). Most QuickCheck implementations use generators in a completely random way, and only report about counterexamples of properties. However, such implementations also provide interface to inspect test data distribution, provided that there is a classification of the domain. For example, after generating 100 integers for testing an integer function, the system can report on the distribution of negative and positive test numbers. This may be of very good use when fine-tuning the probabilities in *frequency*.

Remark (Shrinking.). When the property-based testing finds counterexamples to a property, creates reports about these so that failing cases can be inspected. In practice, however, random data generators may produce incomprehensibly complex and large test cases for disproving a property, and ultimately, such counterexamples hardly help identifying the roots of bugs. To overcome this issue, QuickCheck introduced a technique called test case shrinking: when finding a counterexample, it tries to generate a smaller, yet still failing test case. Shrinking in general is not more than trying to rephrase the counterexample by re-generating parts of it with a decreased generator size, but the system allows experts to define shrinking specifically to a particular data generator, too.

Generating recursive types Generators can be given a name by encapsulating them in functions; thus, recursive generators can be created by means of recursive generator functions. Generators for recursive types (such as syntax trees) need to be designed with care, it has to be ensured that the generator eventually opts for the base case and the branch terminates.

The problem can be handled implicitly or explicitly. One can rely on the already mentioned size attribute of the generators and select one of the base cases when the size is small enough. Alternatively, generators may maintain a recursion limit and implement explicitly bounded recursion; in the method I designed, I used the latter solution via storing this recursion limit in a grammar attribute.

Remark (Lazy generators.). There is a generator that is only important from the technical point of view. Because Erlang is strictly evaluated by its run-time system, there is a dedicated combinator that makes a generator lazy. When writing recursive generator functions, we need to make the evaluation of the generator lazy, by encapsulating the application of the function in an anonymous function. Technically, this job gets easily done with the generator combinator macro called *LAZY*.

Proposition 3.2.2. *QuickCheck data generators define subsets of data types of the programming language, and associate each element (value) with a probability.*

Proposition 3.2.3. *QuickCheck data generators form a monad over the data types of the implementing programming language. The monad operators behave as follows: the* return *operator creates a constant generator, while the binding operators provide successive relation between generation steps.*

3.2.3 Generator for syntax trees

QuickCheck generators can define any language, therefore they can define a programming language (or the syntactic terms thereof) without any problems. Representation is not an issue either: syntax trees can straightforwardly be represented in Erlang by encoding the tree in lists of tagged tuples. The problem of creating a generator for programs boils down to creating a generator for such lists of tagged tuples that faithfully represent well-formed Erlang programs. However, the effort needed to write a generator for a programming language is significant.

Example 3.2.5. Let us consider a simplified example to demonstrate the complexity of writing compound generators. Suppose that we want to define a language that consists of series of expressions represented in a list, where expressions can either bind names, construct literals or compute binary operations (such as addition or multiplication) on the values of other expressions. Furthermore, suppose that the types of the expressions in the list are determined by another list containing type names. Clearly, there are different kinds of information to be shared among the generators for the various syntactic objects, and in general, it can be cumbersome to design the flow of inherited and synthesised data.

Data generators for the above expressions may be scripted in QuickCheck generators as follows. Even for this simple example, the generator functions are hardly readable or extensible, because they contain too much low-level, unnecessary details.

```
exprs([Type|Types] = _Types, Context) ->
  ?LET({Expr, NewContext}, expr(Type, Context),
       [Expr | ?LAZY(exprs(Types, NewContext))].

expr(Type, Context) ->
  oneof([?LET(Name, fresh(Context), {{declare, Name}, Context ++ [Name
      (cont.)]}),
        case Type of
          bool -> ?LET(RandomBool, bool(), {{lit, RandomBool}, Context})
              (cont.);
          int ->
            oneof([
              ?LET(RandomInt, int(), {{lit, RandomInt}, Context}),
              ?LET(VarName, elements(Context), {{var, VarName}, Context
                  (cont.)}),
              ?LET(LeftExpr, ?LAZY(expr(Type,Context)),
              ?LET(RightExpr, ?LAZY(expr(Type,Context)),
                  {{add, LeftExpr, RightExpr}, Context}))
            ])
        end
        ]).
```

When looking at the generator function, it is apparent that a significantly large part of the code is handling context dependencies and passing context. Indeed, in general, complexity of writing generator functions manually is stemming from handling generation order, dependencies among constituents, a number of technical details, and of course, the syntactic noise.

Although the definition of such a generator is inherently imperative, we know that context-free languages, as well as context-sensitive languages, can be specified declaratively by using formal grammars. The above language could be very easily formatted as an attribute grammar in the notation to be presented in the following section.

```
exprs -> {$0.types, ~expr}
expr -> fresh :: {declare, '$1'} [context = '$0'.context ++ ['$1']]
      | when [type == bool] bool                    :: {lit, '$1'}
      | when [type == int ] int                     :: {lit, '$1'}
      | when [type == int ] elements('$0'.context) :: {var, '$1'}
      | when [type == int ] expr expr              :: {add, '$1', '$2'}
```

The declarative notation, as opposed to the imperative one, is definitely more comprehensible, since it is free of the details that are handling control and data flow. As you can observe, structure of data is well embedded into the production rules of the grammar, while information inheritance and information synthesis can be handled concisely with symbol attributes. We are going to exploit this, and build a method that turns grammars to generators.

3.3 Method to grammar-based program generation

The problem of program generation is best solved by employing a formal grammar to synthesise a data generator for syntax trees of well-formed programs. From a declarative, readable description, an imperative generator is created. With this, we solve the problem in two halves: defining the language of interest with a formal grammar, and creating a method that turns the grammar into a data generator. We first investigate the latter issue, then we give a grammar for well-formed Erlang programs.

Generative formal grammars are widely and extensively used for parsing text. Context-free grammars are a great tool for defining the syntax of programming languages, while attribute grammars can express static semantics as well. Most parser combinators (compiler compilers) provide support for some restricted class of attribute grammars; top-down parser generators work well with L-attribute grammars, while bottom-up parser generators usually support S-attributed grammars, for efficiency reasons (no need for additional traversal of the syntax tree). While grammar-based definitions of (programming) languages are mostly used for parsing sentences, they can equally be used for enumerating language sentences.

Notes on attribute grammars. Throughout this section, we assume that the reader is familiar with the basics of formal grammars and languages (including the definitions of formal grammar, generated language, derivation, deterministic grammar and the Chomsky-hierarchy). On the other hand, let us refresh how attribute grammars extend context-free grammars, and what restrictions L and S-attribute grammars add to the general definition.

Definition 3.3.1 (Attribute grammar)**.** An attribute grammar is a context-free grammar augmented with attributes, semantic rules, and conditions. Attributes are attached to grammar symbols, semantic rules compute the values of attributes (possibly in terms of other attributes), while conditions are logical statements on attribute values.

Attributes can be either synthesised or inherited. A synthesised attribute is computed from the values of attributes of the children, therefore carries information upwards in the syntax tree. An inherited attribute at a node in the parse tree is defined using the attribute values at the parent or siblings, thus, it propagates information downwards and across the tree.

Definition 3.3.2 (S-attribute grammar)**.** An attribute grammar is S-attributed, if all attributes are synthesised.

Definition 3.3.3 (L-attribute grammar)**.** An attribute grammar is L-attributed, if each inherited attribute of X_j on the right side of $A \to X_1, X_2, \ldots, X_n$ depends only on the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$ and the inherited attributes of A.

Proposition 3.3.1. *Attribute grammars can define languages that are not context-free, i.e. languages that cannot be defined with context-free grammars.*

Parsing concrete, generating abstract syntax

In static analysis and in compilation, syntactic analysis is based on the concrete syntax of the language, but static semantic analysis is often implemented on an abstract variant of the syntax tree, for the sake of analysis simplicity. In abstract syntax, details not affecting static and dynamic semantics are simply left out. Although the abstract syntax is usually not explicitly defined by a grammar (rather by a data type), the language of abstract syntactic terms can be defined by means of formal grammars, too. Actually, most programming languages have both concrete and abstract syntaxes defined in some way, with the latter leaving out all details of concrete syntax only required for deterministic parsing and code readability.

Parsing. Concrete syntax is almost always formally defined by means of a context-free formal grammar (in BNF) over the lexical layer of the language. Based on this grammar, (automatically synthesised) parsing yields a concrete syntax tree (CST) that can be easily mapped to an abstract syntax tree (AST) representing an abstract term in the language. In this process (Figure 3.1), the CST is implicitly built, while the AST is explicitly created: tokens of the program are processed by parsing in order to recover the concrete syntax tree, then the concrete syntax tree is translated to an abstract syntax tree that is further analysed for static semantic properties.

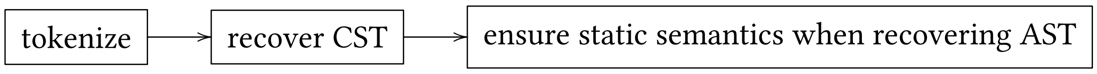


Figure 3.1: CST and AST in program analysis

We say that a parse tree is implicitly built, if there is no corresponding data structure built for the tree, only the call stack of the program maintains the grammar symbols visited. This is the typical implementation of top-down LL parsing via recursive descent analysis, and we will use a very similar technique in case of top-down random generation.

Generation. One could decide to generate random programs based on the concrete syntax grammar of the programming language, but as we aim at generating static semantically valid code, we better build our generation primarily on the abstract syntax. Therefore, we kind of reverse the process used in analysis: we first generate an abstract syntax tree that is correct with respect to the static semantics of the language, and then we explicitly construct a concrete syntax tree that corresponds to the abstract one (there may be many). Observe that the process (Figure 3.2) starts with the AST, so that the grammar we formalise and use for generation is the abstract syntax of the language, rather than the concrete syntax. If the concrete syntax is available as a formal definition yet, some straightforward steps are needed to make it abstract. We will come to this question in the following sections.

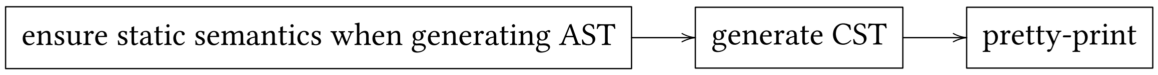


Figure 3.2: CST and AST in program generation

Remark (Grammar ambiguity). When using a formal grammar for parsing, it is a strict requirement that the grammar be deterministic, because the syntax tree recovered for a string has to be unique. However, when using a grammar for sentence generation, it does not have to be deterministic, since the syntax tree is not recovered but built, and every alternative derivation of a particular string is acceptable. Thus, it will not cause any problems that the abstract syntax is not deterministic.

Example 3.3.1. Let us compare the abstract and the concrete syntax definitions of a simple expression language containing integers combined with addition and multiplication operators. The concrete syntax explicitly mentions the symbols sentences are constructed with, including the terminal symbols for addition, multiplication and parentheses.

```
expr -> int
      | expr '+' expr
      | expr '*' expr
      | '(' expr ')'
```

In contrast, the abstract syntax does not include the terminal symbols used in concrete sentences, it associates syntactic constructors instead (enclosed in braces).

```
expr -> {lit} int
      | {add} exp exp
      | {mul} exp exp
```

In the abstract variant, concrete syntactic symbols of arithmetic operations are replaced by labels that carry the meaning of the signs. Also, in concrete syntax, there are no parentheses, since those are only needed to guide the parsing of compound expressions.

Recursive rules. When generating sentences, the choice between the two branches in arbitrary, generation may select the recursive option all the time, resulting in divergence. This termination problem can be tackled by bounded generation, forcing the rewriting system (grammar) to eventually opt for the base case. This can be implemented by using primitive recursion, which keeps decreasing the structural complexity of the branch. In the following sections, we will demonstrate the use of such an explicit recursion bound implemented in attributes to grammar symbols.

```
expr(0) -> {lit} int
expr(n) -> {add} expr(n-1) expr(n-1)
expr(n) -> {mul} expr(n-1) expr(n-1)
```

3.3.1 Generating abstract and concrete syntax

As shown on Figure 3.2, our generation method is two-phased. First we generate an abstract syntax tree, and then we construct a concrete syntax tree that corresponds to the abstract one. In this approach, the grammar-based generation phase takes the abstract syntax into account. How do we define abstracts syntax with a grammar, and how do we connect concrete syntax to it?

Abstract syntax is typically defined as a data type, rather than as a language; as a set it contains terms rather than strings. The specification of abstract syntax greatly varies in the literature, ranging from algebraic data types and constructor signatures to labelled context-free grammars. As written in [64], as mathematical objects, the various abstract syntactic categories are built from aggregations (Cartesian products), alternations (disjoint unions), and list structures. Any notations for these three constructors can serve to define the abstract production rules; in the end, the point is that any element of the abstract language should represent a set of syntactically valid concrete programs.

Example 3.3.2 (Defining abstract syntax)**.** Let us come back to the language defined in Example 3.3.1. The concrete syntax explicitly mentions the signs used to denote the operations ($+$ and $*$), but the abstract syntax only cares about the role of those signs (addition and multiplication). When defining the abstract syntax of this language with function constructors, we would end up with the following three signatures, constructing base and recursive cases, respectively.

$$\begin{aligned} \mathit{lit} &: \mathit{int} \to \mathit{expr} \\ \mathit{add} &: (\mathit{expr}, \mathit{expr}) \to \mathit{expr} \\ \mathit{mul} &: (\mathit{expr}, \mathit{expr}) \to \mathit{expr} \end{aligned}$$

The language of terms producible with the above constructors can be defined with a so-called term-generating grammar, or labelled context-free grammar. The labels/tags attached to the context-free production rules define the abstract syntax constructor to be used on the generated subterms (subtrees), whilst the symbols in the rule determine the type of the constructor. In this document, we prefer defining abstract syntax with such a grammar.

```
expr -> {lit} int
      | {add} exp exp
      | {mul} exp exp
```

Observe that *expr* is a non-terminal, while *int* is a terminal symbol, even though *int* itself is not a 0-ary constructor function in the constructor definition. In practice, *int* will be an already defined syntactic category, with constructors and values being opaque from the grammar point of view.

From abstract to concrete syntax

Once we manage to generate a sentence (or tree) in the abstract syntax, we need to turn it into a concrete sentence (or tree) that can be used for property based testing of the refactoring under test. The mapping between concrete and abstract syntax is not injective: there may be infinitely many different concrete syntax trees that correspond to the very same abstract syntax tree.

Once having an AST, we need to select one out of the set of the possible CSTs the correspond to the AST. The function that maps ASTs to CSTs is usually defined in a recursive fashion, like in syntax-directed translation: by pattern-matching on the abstract object, we construct the concrete object with concrete ingredients given as results of translation of abstract ingredients.

Example 3.3.3. Let us use the grammars defined in Example 3.3.1. By using the abstract syntax, we are able to construct the following abstract object.

```
add(lit(int), lit(int))
```

Now there are several concrete objects that correspond to this very same abstract object, which, for example, may only differ in how they are parenthesised. The following lines show a concrete sentence (determining a concrete tree) each.

```
1 + 2
(1 + 2)
(1) + (2)
```

It is up to the conversion process which concrete object it selects for the abstract object constructed with random generation.

Incorporating synthesised attributes

It is not necessary for us to implement term generation and abstract to concrete object conversion separately. For instance, in compilers it is very common that at the time of implicit construction of the CST, the AST is built simultaneously, explicitly stored in synthesised attributes. We will do the same with the implicit construction of the AST and explicit building of the CST.

Example 3.3.4 (Parsing)**.** In parsing S-attribute grammars, the CST is built based on the context-free part of the syntax, while the AST is accumulated in synthesised attributes at the nodes of the CST — the complete AST is synthesised in the attribute of the root node. This way, there is a derivation-time conversion from concrete to abstract syntax.

```
expr -> int            :: lit($1)
      | expr '+' expr :: add($1, $3)
      | expr '*' expr :: mul($1, $3)
      | '(' expr ')'  :: $2
```

When constructing the abstract syntactic object, symbols connected to parsing and readability (in the concrete syntax) are not included, but they are incorporated in the construction of the abstract object. Labels seen in the previous abstract syntax grammar get recorded in synthesised attributes to hold the necessary information about the categories of subtrees.

Remark (Notation for attribute references). In this section, we will use the attribute access notation introduced in yacc [33]: `$N` denotes the synthesised attribute of the N^{th} symbol on the right hand side of the rule. The synthesised attribute of the LHS is set by the expression following the :: sign (in yacc the LHS is denoted with $$).

Generation. In generation, the above idea of constructing concrete and abstract syntax simultaneously can be used in reverse: we define an abstract syntax grammar, and construct the concrete syntax object in the synthesised attributes explicitly.

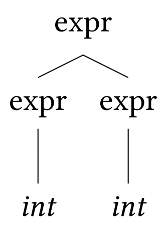
Synthesising text. Let us consider the grammar seen in Example 3.3.1 once more. We can extend it to explicitly generate a text representation of the AST in the synthesised attributes of symbols.

```
expr -> {lit} int       :: to_string($1)
      | {add} expr expr :: "(" ++ $1 ++ "+" ++ $2 ++ ")"
      | {mul} expr expr :: "(" ++ $1 ++ "*" ++ $2 ++ ")"
```

Now this extended grammar can be simplified: the labels can be dropped, as the implicitly built AST is not used in the further process. This formalism demonstrates the true ambiguousness of the grammar we generate with; not only the context-free rules are ambiguous, but there are duplicate-like production rules. Apparently, these are not duplicates, but they belong to abstract term constructors of the very same type signatures (see above, *add* and *mul*). Look at the grammar with the labels removed:

```
expr -> int       :: to_string($1)
      | expr expr :: "(" ++ $1 ++ "+" ++ $2 ++ ")"
      | expr expr :: "(" ++ $1 ++ "*" ++ $2 ++ ")"
```

The following abstract tree could be generated with the grammar:

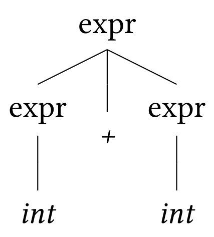


This abstract tree derived with the above grammar tells nothing about the operation used; it may be either addition or multiplication. However, a corresponding explicitly generated concrete text clarifies, which is synthesised in the semantic attribute of the root *expr*.

Synthesising tree. It is up to the grammar writer, what kind of concrete object the grammar synthesises explicitly. In practice, we do not generate strings, but we construct a CST. For example, in Erlang, we may produce a syntax tree represented as tagged tuples and lists. The previous text-generating grammar can be phrased to generate trees instead:

```
expr -> int       :: {expr, $1}
      | expr expr :: {expr, $1, '+', $2}
      | expr expr :: {expr, $1, '*', $2}
```

The only difference is in the computation of the synthesised attribute. For the above drawn, implicitly generated abstract tree, the following concrete tree could be generated explicitly:



3.3.2 Generator language based on L-attribute grammars

In this section, we overview the L-attribute grammar based data generator language that we specifically designed for random program generation. We show a simple example and list some language features that support easy and concise generator definition; more detailed description of the formalism can be found in [15].

The brief informal semantics of the various language features, along with some examples, are provided in this section, while the next section gives more precise meaning by explaining the conversion of the grammar into generator functions.

Example 3.3.5 (Definition for the $a^nb^nc^n$ context-sensitive language)**.** To give a quick and fairly complete idea of what the language looks like, let us specify an L-attribute grammar that generates the well-known context-sensitive language $\{a^nb^nc^n \mid n \in \mathbb{N}\}$ over the alphabet $\{a, b, c\}$.

In this example, the mapping between abstract syntax and concrete syntax is very simple: the concrete syntax is a string that contains the generated alphabet symbols, that is, the terminals read from left to right.

It is the root symbol that generates the entire sentence in its synthesised *value* attribute, while the other non-terminal symbols synthesise sub-languages. The context-sensitive properties are ensured by the synthesised attributes of `a_seq` and by the inherited attributes of `b_seq` and `c_seq` (all named *size*), based on which the repetition construct makes sure the generated strings contain equal number of a, b and c letters. In the traditional attribute grammar definition of this language, attribute conditions are employed to ensure this context-sensitive property.

```
Nonterminals abc_seq a_seq b_seq c_seq.
Terminals a b c.
Rootsymbol abc_seq.

abc_seq -> a_seq
        ~> b_seq [ @size = '$1'.size ]
           c_seq [ @size = '$1'.size ]
        :: '$1' ++ '$2' ++ '$3'.
a_seq -> {a} :: '$1' [ @size = length('$1') ].
b_seq -> {'$0'.size, b}.
c_seq -> {'$0'.size, c}.

Erlang code.
a() -> 'a'.
b() -> 'b'.
c() -> 'c'.
```

Listing 3.1: Definition of the language $a^n b^n c^n$

Listing 3.1 shows the concrete syntax we use to script L-attribute grammars. The description format is very similar to grammar specifications in commonly used parser combinators (e.g. yacc or yecc): it declares the grammar symbols in directives, then enumerates context-free production rules, and attribute computations are given in the host language. Nevertheless, we are going to overview how different this language is from traditional S-attribute grammar based parser languages.

The value attribute. Our attribute grammars are not as restrictive as those in parser generators: symbols in our definitions can have multiple synthesised and inherited attributes. However, there is a restriction: each symbol must have a synthesised attribute called *value*, which holds the explicitly built concrete syntactic object belonging to the symbol.

For terminals, the generator function does nothing but creating this synthesised attribute by generating an element of the type or syntactic category associated (e.g. *int* or *var*). For non-terminals, following the productions rules, the symbol :: introduces the semantic routine calculating the distinguished synthesised attribute. When such a clause is not provided, the system aggregates the value attributes generated by the right hand side of the rule.

Attribute manipulation. Attribute computations are coupled with the symbols they belong to, which allows for static (lexical) scoping of attributes. When setting an attribute by its name, the symbol it belongs to does not have to be explicitly selected; attribute modifications are always performed on the symbol they follow. This enables easily comprehensible attribute computations and attribute dependencies are solved by the lexical scoping (due to the fact that right hand side symbols are generated from left to right).

Syntactically, an attribute setting clause can follow either of the right hand symbols (resulting in setting inherited attributes to RHS symbols) or can be placed after the entire rule (setting synthesised attributes to the LHS symbol). For technical reasons, the attribute name is preceded by an at sign (@). When accessing attributes, we follow the traditional formalism introduced in yacc [33]: symbols are referred to by their position in production rule (e.g. '$2'.foo refers to the attribute named *foo* belonging to the second RHS symbol, whilst '$0' refers to the LHS).

Conditions. Unlike in S-attributed grammar formalisms used in widely used parser generators, this language supports explicit conditions attached to production rule alternatives, behaving like guards. (These conditions are slightly different from the general definition, because our guards can only refer to the attributes of the LHS.) Following the *when* keyword, a boolean expression on the inherited attributes of the LHS can be used to guard the rule alternative. If all rule alternatives are guarded and none of them are applicable, the generation stops with an error.

Example 3.3.6. Let us rephrase the rule belonging to *b_seq* in Listing 3.1. We replace the repetitive generation with recursion, and introduce a guard for the base case.

```
b_seq -> (when [@size == 1]) b
       | b b_seq [ @size = '$0'.size-1 ].
```

With the guard, the first rule alternative can only be selected by the generator function if the inherited size attribute equals one.

EBNF-like notations In a large number of cases, recursive production rules are simply employed to produce a series of symbols. In commonly used parser generator formalisms based on BNF, extensive use of such recursive rules lead to incomprehensible grammars. This issue is tackled in EBNF by introducing repetition: symbols enclosed in braces.

```
<exprs> ::= <expr> <exprs> | <expr>     (BNF)

exprs = {expr}                          (EBNF)
```

We employ this simplification in our generator language, in enhanced versions as well.

```
exprs -> { expr }
exprs -> { P, expr }
exprs -> { ~ expr }
```

The first line generates some expressions, the second line generates P expressions, while the third line generates some expressions dependently, such that they inherit each other's attributes. Dependent generation of series of symbols will be of good use when handling variable and function name context in attributes.

The repetition construct is implemented as recursive, so termination has to be guaranteed during generation. The language allows the repetition to explicitly define the number of symbols to be generated, but if the is no quantity specified, the system generates an upper bound before performing repetition. Furthermore, the repetition can be bounded by the size of another list, in which case the system automatically passes the list values to the generated symbols, providing a foreach-like behaviour to this language construct. By default, the repeated symbols are generated simultaneously, adding the $\sim$ symbol makes their generation dependent. The next section will elaborate more on how recursive generation is bounded.

Probabilities Last but not least, a generation-specific feature in the language is weights attached to rules, which make the described grammar stochastic: all rules get associated with a probability calculated as dividing their specified weight by the sum of weights of all alternatives.

```
a_seq -> (*2) a       [ @size = 1 ]
       | (*1) a a_seq [ @size = length('$2')+1 ].
```

With this feature, we get a control on which sub-languages are included in the generated sentences with a higher probability, therefore influencing test case distribution.

3.3.3 The synthesised data generator

Semantics of stochastic L-attribute grammars given in the above described language could be handled formally, without even connecting them to generators. We do not define the denotational semantics of the generator language, but describe a translation-like semantics that maps L-attribute grammars to functions written with QuickCheck generators in Erlang. In particular, a generator function is synthesised for each and every non-terminal symbol in the grammar, which can be used to generate the sub-languages.

The translation algorithm

In this section, we overview the algorithm that translates an L-attribute grammar expressed in our notation to a data generator (more precisely, a series of functions defining generators). We give a high-level explanation supported with examples, but we note that in the actual implementation there are various optimisations that make the generator code hard to comprehend.

Remark. When synthesising the generator, we assume that the attribute grammar defined with the generator language is reduced and well-formed. In case of violation of these properties, the generated Erlang/QuickCheck code may contain syntactic or static semantics errors.

Synthesis associates a data generator function with each and every symbol in the grammar. These functions have one formal argument, the set of inherited attributes to the symbol, and their return value determines the synthesised attributes of the symbol. The only exception is that the root symbol does not take any arguments as it cannot inherit attributes. Generators for terminals should already be defined in the host language (either user-defined or pre-defined for basic types); terminals are mapped to the corresponding generator functions by their name. The following main steps are performed in the translation:

- Generate the header of the Erlang file along with the attribute-manipulation primitives (for reading and writing attributes of symbols);
- Output the grammar-specific higher-order generator functions (such as those that generate repetitions of symbols);
- Process non-terminals one by one consecutively, and synthesise a generator function for each.

In the rest of this section, we explain how the different parts of the grammar definition are translated into (elements of) generator functions.

Non-terminals

Apparently, the most complex part of the translation is how the non-terminals are generated. When processing non-terminals, all the production rules are considered that have the symbol in question on their left hand side (LHS). If there are multiple alternatives, they are combined with either the *oneof* or *frequency* generators, depending on whether the alternatives are weighted with probabilities. This way, the sub-languages generated by the alternatives are unified (see Section 3.2.2).

Example 3.3.7. A grammar defining a non-terminal with two one-symbol alternatives is translated to a function relying on the *oneof* generator combinator.

```
a -> b | c
```

The resulting generator function:

```
a(Attrs) -> oneof ( [ b(Attrs) , c(Attrs) ] ).
```

Grammar rules

How do we generate the right hand side (RHS) of a rule? Symbols are mapped to tuple generators that call the appropriate generator functions associated with the symbols of the RHS. When there are no dependencies, the RHS generators can be executed simultaneously, provided that the attached attribute computations pure (do not have any side-effects).

Example 3.3.8. Consider an example defining a non-terminal with one alternative containing two symbols.

```
a -> b c
```

The non-terminal (and the corresponding rule) is mapped to a generator function:

```
a(Attrs) -> merge({ b(Attrs) , c(Attrs) }).
```

We address attribute merge and aggregation in the subsequent paragraphs.

When there is dependency between the siblings on the RHS, dependent generation is applied by using the *bind* generator combinator (see Section 3.2.2).

Example 3.3.9. Complicating the above example a bit, we can introduce dependent generation.

```
a -> b ~> c
```

The non-terminal (and the corresponding rule) are mapped to the following generator function:

```
a(Attrs) -> ?LET(Attrs2, b(Attrs), c(Attrs2)).
```

Note that top-down, left-to-right generation is ensured by the host language evaluation strategy. As the RHS symbols are generated as a tuple, they are generated in a left-to-right order because of the strict evaluation strategy in Erlang. With the symbols mapped to functions, symbol stack managed by the run-time stack, and visit order provided by the host language evaluation strategy, our synthesised generation algorithm generation mimics recursive descent parsing.

Remark. The grammar compiler does not implement a complete recursion analysis on the production rules, so recursively generated symbols have to be marked in the header of the grammar. Such symbols are encapsulated in a *lazy* generator, to 'disable' strict evaluation of Erlang, thus avoiding divergence.

Attributes

Attributes are stored in local variables of generator functions, represented as a list of key-value pairs. Scoping of attributes this way is completely managed by the host language, Erlang, via variable scopes. Attribute modification and access is solved via the operations available in Erlang for key-value lists.

There are no assumptions on the attribute computations written in the host language, the snippets are simply encapsulated into begin-end blocks. The computed attributes are stored in variables and are passed to attribute modifier macros.

Conditions

Conditions attached to rule alternatives are used to evaluate whether a particular alternative is subject to application or would result in an invalid generated sentence. The conditions are evaluated at the beginning of the generator function, based on the actual inherited attributes of the LHS; alternatives with falsified conditions are not included in the current generation.

Parameters

In our language, non-terminal symbols can have parameters. These parameters are simply mapped to parameters of the generator function, therefore, pattern matching in the host language will do the selection between the alternatives.

Example 3.3.10. Consider a non-terminal taking two parameters.

```
a(N, 1) -> b c
```

The resulting generator function takes these parameters in addition to the attribute set. When a appears on the right hand side, actual parameters have to be supplied.

```
a(Attrs, N, 1) -> merge({ b(Attrs) , c(Attrs) }).
```

Although a similar behaviour could be achieved with inherited attributes and conditions thereon, sometimes it is more effective to employ the pattern matching available in the host language. We exclusively use this feature when composing recursive generators from recursive production rules.

Repetition

Different forms of repetition are handled differently. If a symbol is simply put into braces, multiple instances of the symbol are generated with the *list* generator combinator. If there is a quantity specified (i.e. exact number of symbol instances), the *vector* combinator is used as a wrapper on the generator function of the symbol.

When dependent generation is used, the symbol instances are generated consecutively, exploiting the *bind* combinator to pass the synthesised attributes of a node to its right siblings. This latter functionality is hidden behind a function called *dependent_list*, which takes the number of instances needed, the generator function, and the attributes to start with, and hides recursion and attribute handling.

Attribute aggregation. The repetition in the rule is treated as an aggregated, standalone RHS element. Since each and every symbol instance in the repetition may synthesise attributes with the same name, attributes have to be aggregated into one that will belong to the repetition construct. In the simplest case, this may be putting the values into a list, or if they are lists themselves, concatenating them. However, different attributes require different aggregation, so beside having some default aggregation operations, the creator of the grammar can specify how different attributes are aggregated or merged, based on their name and type.

Example 3.3.11. Consider the generator for a terminal symbols seen in Example 3.3.5 with a specified quantity (namely, 3):

```
a_seq -> {3, a} :: '$1'.value     -->     "aaa"
```

The entire repetition gets index 1, whilst its value attribute ('$1'.value) is the aggregation of the values of the several a terminals (the list of characters becomes a string).

Attribute splitting. While synthesised attributes of instances need to be combined together, inherited attributes of the repetition also need special care. Beside normal inheritance of attributes to each symbol instance, in case of foreach-like constructs, the parameter list is split and each symbol instance inherits an element from the list in an attribute called *param*.

Note that the recursion that implements these generators is always bounded, there can be no divergent generation paths stemming from the repetition construct.

Implicit inheritance in dependent generation

Although inheritance is made automatic from parents to children, and in dependent sibling generation, not necessarily all attributes are passed over, partly for memory-efficiency reasons, and partly to avoid name conflicts.

Whoever writing the grammar can define a function called *inherit*, which takes a list of attributes and can filter it based on names, types or content. With this, commonly inherited attributes can be implicitly inherited, without explicit attribute computation attached to production rules. This proves to be a useful feature when passing e.g. bound variable names from symbol to symbol.

Example 3.3.12. Consider our original example grammar:

```
abc_seq -> a_seq ~> b_seq [@size = '$1'.size] c_seq [@size = '$1'.size]
```

With the implicit inheritance of the attribute named *size* the above line simplified to:

```
abc_seq -> a_seq ~> b_seq c_seq
```

Note that both `b_seq` and `c_seq` implicitly inherit the synthesised attribute of `a_seq`. Dependence is not needed between `b_seq` and `c_seq`, because the latter does not refer to any synthesised attributes of the former.

Proposition 3.3.2. *The language generated by context-free part of a grammar given in the EBNF-free formalism is the same as the language implicitly constructed by the generator synthesised for the context-free part of the grammar.*

Proof. The synthesised generator functions implement a recursive descent algorithm, which simulates a push-down automaton that is equivalent to the grammar. Calls to terminal generators (i.e. terminal names in the possible call chains) represent the sentences in the language accepted by the automaton, which is the same as the language generated by the grammar. The only difference from a recursive-descent parser algorithm is that in the automaton implemented with generator functions, alternatives are encoded as epsilon-transitions without any conditions. □

Remark. In recursive descent syntactic analysis, terminals are 'accepted' by the algorithm, while in recursive descent analysis, terminals are 'instantiated'. However, the possible activation records in the stack (defining the sentential forms of the grammar) are the same.

A generated generator

Here we briefly examine the generator functions synthesised from the grammar presented in Example 3.3.5. The following code has been generated with the grammar compiler and it contains a fairly large amount of syntactic noise and apparently useless variables. The generator generator makes sure that the generated code faithfully implements the grammar, handles the outputted snippets in a generic way, and it does not do any simplifications on the synthesised code. The synthesised code is not subject to reading, but in the following, we explain the elements of it.

```
abc_seq() ->
  V0 = [],
  ?LET(V1, (a_seq(inherit(V0))),
    begin
    S1 = inherit(V1),
    ?LET(
      {V2, V3},
      {b_seq(?setattr(size, begin ?getattr(size, V1) end, inherit(S1))))
          (cont.),
      c_seq(?setattr(size, begin ?getattr(size, V1) end, inherit(S1))))}
          (cont.),
        begin
          R2 = begin (?getattr(value, V1)) ++ (?getattr(value, V2))
                       ++ (?getattr(value, V3)) end,
          ?setattr(value, begin R2 end,
                    (inherit(aggregate({inherit(V2), inherit(V3)}))))
        end)
  end).
```

In the grammar, `abc_seq` was marked as the root symbol, so it does not take any arguments (meaning it cannot have any inherited attributes) and therefore can be used as a standalone data generator in QuickCheck. The attribute set is initialised as an empty list (line 2).

In the body of the function, first the `a_seq` generator is applied to generate its synthesised attributes (`V1`, line 3), which are passed to the tuple generator synthesised from the symbols `b_seq` and `c_seq` by using *bind* (used via the macro `LET`, lines 7-9), because of the dependency modified. The symbols `b_seq` and `c_seq` are generated independently of each other, the synthesised attributes of `b_seq` are not passed to `c_seq`, but they both get their *size* attribute set set to the *size* attribute of `a_seq` (lines 8-9). After all symbols generated, the value of the LHS is computed and stored in `R2` (lines 11-12), which is set as the synthesised value attribute of the LHS (lines 13-14).

```
1  a_seq(V0) ->
2    ?LET(V1,
3      (independent_list(
4        fun (Attrs) -> ?terminal((a()), Attrs) end, inherit(V0)))),
5      begin
6        R1 = begin ?getattr(value, V1) end,
7        ?setattr(value, begin R1 end,
8                 (?setattr(size, begin length(?getattr(value, V1)) end,
9                   (inherit(V1)))))
10     end).
```

The synthesised function for `a_seq` takes as argument the attributes it inherits (`V0` (line 1), then generates a number of a terminals (lines 3-4); the updated attribute set is stored in `V1`. The value attribute gets saved in `R1` (line 6), and last but not least, the final attribute set is updated so that the *size* is set to the length of the value attribute (lines 8-9).

```
1  b_seq(V0) -> independent_list(?getattr(size, V0),
2                 fun (Attrs) -> ?terminal((b()), Attrs) end, inherit(V0)).
3
4  c_seq(V0) -> independent_list(?getattr(size, V0),
5                 fun (Attrs) -> ?terminal((c()), Attrs) end, inherit(V0)).
```

The functions synthesised for `b_seq` and `c_seq` are very similar due to the similarity in the corresponding grammar rules. They generate terminals (b and c, respectively) by using the independent list generator combinator we defined before.

There are some macros and functions used in the above code that are used for handling attributes. Here we give a brief explanation for those:

- *getattr*: given an attribute set and an attribute name, returns the value of the attribute;
- *setattr*: given an attribute set, a name and a value, returns an updated attribute set;
- *terminal*: sets the synthesised value attribute to the terminal to the value it generates;
- *inherit*: filter function for implicitly inherited attributes;
- *aggregate*: aggregation of synthesised attributes.

Implementation

The above described grammar language has been implemented with a compiler that translates grammar files to Erlang modules containing QuickCheck data generators for non-terminal symbols. The grammar language literally embeds Erlang code, and therefore in semantic routines any Erlang code can be written. The semantics of the grammar is in many aspects determined by the semantics of Erlang, including evaluation order and scoping rules. The implementation of the translation has been used to transform the following grammar (defining Erlang) to the corresponding random data generator.

3.4 An attributed grammar for Erlang

Proposition 3.4.1. *Erlang is not context-free, therefore it cannot be defined with a context-free grammar.*

Proof. The lemma can be proved by using the pumping lemma for context-free languages. □

As Proposition 3.4.1 suggests, we cannot specify a context-free grammar to define Erlang. At the same time, Proposition 3.3.1 tells us that we can define it with an attribute grammar, which means we can express the syntax and static semantics of Erlang in our grammar-based generator language. In this section, we overview some details of the L-attribute grammar we defined for Erlang, the grammar we use for random generation of programs fed into refactoring property checks. The section only quotes some snippets for explanation, but the entire definition is available online [31].

By using both inherited and synthesised attributes, we can specify context-sensitive aspects (static semantics) of Erlang, meaning we can ensure properties such that the program does not refer to unbound variable or function names. Thus, the generated programs will pass compilation, which is important, as it would not make sense to generate programs that are inherently invalid for testing refactoring engines.

Syntax

Static semantics is defined over abstract syntax, so first of all, we need a syntax definition. We designed our grammar formalism to be similar to yecc [33] on purpose: this way the already available concrete syntax definition can be reused, at least partially. Turning a concrete syntax definition to abstract syntax is pretty straightforward in many cases, because the only task is to associate terminal symbols with abstract elements and labels. For instance, consider function definitions:

```
function_clauses -> function_clause ';' function_clauses
                  | function_clause '.'
function_clause -> '(' patterns ')' '->' expressions
```

By getting rid of the connective, concrete syntax terminal symbols (quoted ones), we transform the concrete description to an abstract one. We omit labels in this example, and use the EBNF notation for repetition.

```
function_clauses -> { function_clause }
function_clause  -> patterns expressions
```

With such changes in the concrete syntax grammar, we can easily reuse most of the official context-free grammar defined for Erlang, which saves a lot of time.

Syntactic categories. The grammar defines a number of syntactic categories. Most importantly, it generates a list of files, which may be either modules or headers. In generated modules, the grammar builds an arbitrary number of function definitions, which may have one or more function clauses. Headers are generated to contain macro and type definitions. To support function body generation, there are separate syntactic categories for patterns and expressions of various types. If the type is specified in an inherited attribute, the non-terminals generate a pattern or expression with the specified type, otherwise a random type is synthesised and being used.

Static semantics

Syntax is easily put in context-free rules, static semantics is more interesting to specify. In the following, we explain how we ensure some important static semantic properties of Erlang programs via attribute computations and conditions on the attributes.

Function name scope. Function name scope, unlike in C and C++, is not lexical. In order to make all functions be able to refer to each other (including the first one calling the last one in order), function specifications are generated before the actual module code generation. The synthesised function dictionary (containing module and type information of all functions to be generated) is passed to all elements generated, it is an automatically inherited attribute.

Example 3.4.1. The function dictionary enumerates all functions with the module they should be generated into, with their name and type:

```
[{module1, function1, 1, [int()], int()},
 {module1, function2, 2, [int(),bool()], char()}
 {module2, function1, 1, [int()], int()}
]
```

When the *functions* non-terminal is used to actually generate the functions from the descriptions, it iterates through the dictionary. Apparently, for one particular module, only its local functions are generated, but all the function specifications are passed in an inherited attribute for the grammar to be able to generate inter-module function calls.

```
module ->
  :: ModName = '$0'.param, AllFuns = '$0'.allfuns
  ~> functions [@allfuns=AllFuns, @module = ModName,
               @modfuns=[F || F<-AllFuns,{MN,_,_,_}<-[F],MN==ModName]]
  :: {Funs, _FunInfos} = lists:unzip('$1'),
     ?S:form_list([?S:attribute(?S:atom(module), [?S:atom(ModName)]),
                   ?S:attribute(?S:atom(compile), [?S:atom(export_all)])
                   | Funs]).

functions -> {'$0'.modfuns, function} [@funs = '$0'.allfuns].
```

Based on the pre-generated function specifications, a number of modules with their functions are generated. When generating function references, or function calls in particular, only those signatures can be used for generation that are contained in the pre-generated set. Thus, provided that all the pre-generated signatures are defined by the generation, only referring to these signatures cannot result in invalid function references, avoiding related static semantics and compilation errors.

Example 3.4.2. The function generator takes into account the name, the arity, the argument types and the return type of the function and generates accordingly.

```
function ->
    :: {_Mod, Name, Arity, {Types, RetType}} = elem('$0'.param),
       %% Non-recursive funs
       Funs = [{M, N, A, {T, RT}} || {M, N, A, {T, RT}} <- '$0'.funs,
                N =/= Name orelse A =/= Arity orelse M =/= '$0'.module]
    ~> {?ClausesPerFunction, funclause}
       [@types = Types, @rettype = RetType, @funs = Funs]
       last_funclause
       [@types = Types, @rettype = RetType, @funs = Funs]
    :: {?S:function(?S:atom(Name), filter_clauses('$1') ++ ['$2']),
        elem('$0'.param)}
       [@vars = [], @value = []].
```

Remark. In parser generators, similar information is stored in symbol tables, which are implemented as global variables. In our solution, these details are kept in the scope of the attribute grammar; not only the information is passed in attributes, but the signatures themselves are generated as synthesised attributes of dedicated symbols.

Variable name scope. In Erlang, variables in a clause exist between the point where the variable is first bound and the last reference to the variable in the clause. Unbound variable names are not allowed to be used in expressions.

In our grammar, variable names are not pre-generated, but the variable context is dynamically extended with bindings generated, and passed to subsequent expressions. By only choosing variable names from the current active context, it is guaranteed that the generated code will not refer to unbound names, thus avoiding related static semantic errors.

Example 3.4.3 (Making use of dependent generation)**.** Let us consider a simplified rule that generates a clause of a function. A clause consists of some patterns (matching the formal parameters) and a body (series of expressions). Obviously, during generation, we want the function body to refer to the variables in the pattern (the function signature), so the variables generated there should be visible when generating the function body.

```
function_clause -> {~ N, pattern}
                ~> {~ M, expr}
                :: create_clause('$1', '$2').
```

Supposed that the variable environment is an automatically inherited attribute (as it is in our grammar), dependent generation of the patterns and the body expressions (the ~> sign) ensures the required connection between the head and the body of the clause.

Extended static semantics

As mentioned above, pre-generated function descriptions contain randomly generated type information; nevertheless, this would not be necessary for generating compilable programs. Type-correctness is part of the so-called extended static semantics we introduced, which induces the concept of well-formed programs.

Erlang is dynamically typed. Ill-typed expressions can be compiled without any errors, since type-checking is done dynamically, at run-time. However, those programs that run into silly run-time errors due to the wrongly typed functions or expressions are waste of time from the point of dynamic verification. When comparing functions before and after refactoring, it is a beneficial if they have a meaningful observable behaviour: they return some values, have some side-effects, which we can compare. Therefore, the static semantics is extended so that it defines the language of well-typed (well-formed) Erlang programs. Types are generated by ordinary production rules in the grammar, just like other syntactic elements, having their own abstract representation (we reused the official type representation used in the Erlang compiler).

Example 3.4.4. The language of expressions is the union of expressions of different types, and can take into account if its type has already been generated and is passed as an inherited attribute. Also, generated expressions synthesise their type, which can be used in subsequent phases, if needed.

```
expr(N) -> when subtype_of(integer) integer_expr(N)
         | when subtype_of(float)   float_expr(N)
         | when subtype_of(boolean) boolean_expr(N)
         | when subtype_of(atom)    atom_expr(N)
         | when subtype_of(string)  string_expr(N)
         | when subtype_of(list)    list_expr(N)
         | when subtype_of(tuple)   tuple_expr(N).

integer_expr(N) -> ... [@type = ?T:t_integer()]
```

3.5 Checking Erlang analysis and refactoring with PBT

Having a QuickCheck generator synthesised for the Erlang programming language, we can use it to construct random programs for testing. Here we make extra benefit of expressing the semantics of the grammar in terms of QuickCheck generators, because the language defined by the grammar is turned into a full-featured data generator. Namely, our synthesised functions are more than just random generators. Not only we can produce programs with them, but the original grammar definition seamlessly integrates into the QuickCheck system: the grammar symbols get inherently sized, and sub-languages do have shrinking functionality. When composing properties on the generator synthesised from the grammar, QuickCheck will be able to control the complexity of the generated programs and simplify counterexamples automatically.

In this section, we overview the use cases in which we applied the random program generator for testing analysis and transformation of Erlang programs. The first use case is about testing refactoring transformations, while the second use case addresses testing of data-flow analysis and graph consistency.

Program equivalence

At the beginning of the chapter, the problem statement considered a property that expresses the correctness of refactoring. The property, expressed in a first-order logic formula, relies on a predicate that tells whether two programs are semantically equivalent.

$$\forall p \in \mathit{WellFormedProgram} : \mathit{equivalent}(p, \mathit{refactor}(p))$$

Semantic equivalence can be defined in multiple different ways for programs written in various paradigms. For Erlang programs, we decided to design a simple notion of equivalence based on observable behaviour. Although Erlang is functional, it is not pure, expressions can have side-effects. We define two functions equivalent if and only if they produce the same output value and same side-effects for a particular input value. Two modules are defined equivalent if their functions are equivalent. With our definition of equivalence, only functional behaviour is considered, real-time temporal constraints or memory constrains are not necessarily preserved during a transformation.

In order to perform property-based testing of refactoring, we generate a large number of programs with our synthesised generator, along with randomly generated refactoring commands on them. After the refactoring transformation has been executed, all functions are inspected and tested for equivalence with a number of random input values generated based on their type signature. Execution of functions on the random input values can be done in parallel.

Example 3.5.1 (Identifying wrong refactoring)**.** Suppose that the following program is generated by the generator for testing the refactoring "generalise function" (for the definition of the refactoring, see Section 4.4[3].

```
helloworld() -> f("world"). %% writes hello world
hello(Name)  -> write("hello "), write(Name).
```

Also, suppose that the refactoring transformation, by accident, fails to enclose the extracted expression in an anonymous function. Due to the strict evaluation strategy in Erlang, the execution of the output of "hello" and "world" get swapped, meaning the order of side effects have been changed. This signals that the refactoring was incorrectly performed and the behaviour is not preserved.

```
helloworld()   -> f("world"). %% writes worldhello
hello(Name)    -> hello(Name, write(Name)).
hello(Name, X) -> write("hello "), X.
```

Our equivalence checking system executes all functions multiple times (with randomly generated arguments) to check if they behave the same way as before the transformation. Identifying that the side-effects executed by the function *helloworld()* happen in different order before and after the transformation, it can conclude that the refactoring was incorrect, the test fails and the counterexample is presented to the programmer.

Analysis specifications

Another testing we implemented by using the random generator for programs is checking of program representations built by static analysers. In particular, we implemented a specification-based validity (consistency) check on the semantic program graph built in our Erlang static analyser tool.

$$\forall p \in \mathit{WellFormedProgram} : \mathit{valid}(\mathit{analyse}(p))$$

The testing uses the same program generator that we used for checking refactoring correctness. However, the random programs are not transformed, only analysis is performed on them, which turns the program text into a graph model.

The randomly generated, complicated, sometime nonsense, programs are fed into the static analyser and the resulting model is checked for consistency. The consistency property is defined in terms of syntactic and semantic relationships of objects in the program. An example mentioned in the paper we published on this topic explains the validity property of data-flow graphs built on syntax trees. Even though the incremental data-flow analysis implementation is pretty complicated, the definition of the data-flow relation can be boiled down to a rather simple property to be checked.

[3]For the sake of simplicity, we use a function called *write* to emit side effects in this example. The current grammar would generate *io:format* calls, which would have a similar effect.

3.6 Summary

In this chapter, I gave an overview on a method for property-based testing of refactoring engines with randomly generated source code. The generic approach I designed is based on L-attributed grammars, which are given a meaning by transformation to the language of QuickCheck generators. This conversion turns an inherently declarative, easily comprehensible definition into an imperative implementation. As a result, sentences of any language defined with an L-Attribute grammar can be sampled, and shrunk in property-based test properties. In my particular case study, the Erlang programming language was formalised with a grammar to enable random generation of well-formed Erlang programs.

Thesis 1. *I have developed a method for transforming stochastic L-attributed grammars into QuickCheck data generators. I have composed an L-attributed grammar for a sub-language of Erlang and used the previous method to synthesise a data generator for well-formed programs. By using this generator, analysis and transformation implementations have been verified on randomly generated, semantically valid programs.*

This chapter discussed dynamic verification of refactoring transformation; the following chapter addresses static, formal verification thereof.

Future work

There is room for improvement in both the accurateness of the language definition as well as the way the grammar is utilised for randomised program generation.

As for the first aspect, the attribute grammar for Erlang could be extended to accommodate some specifications associated with the transformation the generated programs are supposed to test, conceivably in a modular way. It would be worth investigating the random generation of programs that comply with some form of a functional specification.

Considering the second aspect, definitely complete random generation should be replaced with iterative or adaptive sampling, in order to increase the usefulness of test cases. Another option could be testing with bounded exhaustive generation. In the end, the goal is to increase coverage, both in the tested code and in the equivalence classes in the generation domain.

4 Verifiable and executable specification of refactoring

"A language that doesn't affect the way you think about programming is not worth knowing." (Alan J. Perlis)

Refactorings can be defined in various formats at various abstraction levels. Some widely referred refactoring catalogues use informal, English explanations supported by some simple examples, while academy keeps looking for mathematically precise definition methods. Although informally described refactorings are easily readable and pretty much good for end-users, they are inherently ambiguous and cannot be interpreted in a formal system. The other typical level of refactoring specification is the syntax tree manipulating algorithms implemented in refactoring tools. Obviously, these contain all details for the refactoring to be executable, but they are expressed in a form that is not amenable to formal verification.

Widely used refactorings all mostly given in high-level, general purpose programming languages. Unfortunately, it is impractical to verify implementations written in high-level languages (such as C++, Java or Erlang) because those are too complex to have tractable formal semantics definitions (i.e. definitions which can be reasoned about easily), making it impossible to rigorously prove properties about designs expressed in them. Yet, sometimes more correctness guarantees are of demand than that of dynamic verification can provide, so we need to find a specification method that supports formal verification and is executable at the same time.

Refactoring tools are complex, multi-tier software: to ensure a refactoring correct, one needs to prove all the components correct, from the highest level of transformation logic to the lowest level of removing or adding nodes to the syntax tree of the program. There are no theoretical obstacles of formal verification of large-scale software, it just takes excessive amounts of work and expertise. Full-stack verification of refactoring is not a realistic goal (at least for this dissertation), but a high level of assurance may be achieved by formally verifying the implementations on the model level, treating the underlying analysis and model manipulation as trusted components. We narrow down the focus by capturing and verifying refactoring on the model-to-model level.

Problem statement

Program transformations and refactorings are understood as algorithms taking source code and returning source code, but for practical reasons, this process is usually divided into three phases: analysis, transformation and synthesis. Syntactic and semantic analysis turns source code into a complex program model, the model is altered by transformations, and finally, synthesis converts the model back to source code. If we treat analysis and synthesis as trusted, the transformation logic to be verified is an algorithm on the model level. Still, even on the model level, manually verifying tens of transformations would be pretty challenging, not mentioning that some refactoring systems are partly extensible, such that custom transformations can be added. Cannot we automate the verification of transformation definitions somehow?

The difficulty of verification depends on the abstraction level on which the transformation is defined. Low-level implementations are hard to be verified, because they contain too many details, while high-level specifications do not define the transformation fine-grained enough. Our main point in addressing this problem is the following: all refactorings can be expressed as graph transformations, but not all graph transformations are refactorings. Therefore, we must focus on expressing a restricted set of transformations on a level of abstraction that is in between the corner cases mentioned above. We need to design a language that only allows for defining a restricted set of graph transformations, those that define behaviour-preserving modifications. Obviously, at the same time it should not be too restricted, because it has to be able to express a fair amount of real-world, useful refactoring transformations.

We need to design the proper abstraction level for defining refactoring, which is both executable and verifiable. The refactoring specification has to be low-level enough to be interpretable as an algorithm or function that maps program models to program models, while at the same time it has to be high-level enough to provide readability and verifiability with a reasonable amount of effort. We aim at creating a system in which refactorings are programmable, executable, and semi-automatically verifiable for correctness.

Structure of the chapter

This chapter is structured as follows. We first summarise the related work in the field of transformation formalisms and verifiable refactoring definitions in Section 4.1, then we overview term rewriting and strategic rewriting in Section 4.2 to prepare the presentation of the language we developed based on term rewriting (Section 4.3). In order to demonstrate the applicability of the method, we present a complex case study refactoring definition in Section 4.4. Finally, we identify limitations and future work in Section 4.5.

4.1 Related work

There are tons of ways to represent programs, define semantics of models, their transformations, as well as the semantic equivalence requirement. In this section, we overview the most important and influencing approaches to executable and verifiable refactoring specification.

We note that although the abstractions for defining refactoring, with the levels of correctness guarantees, are varying from approach to approach, almost every one of them incorporates the fundamental work of Opdyke [57] that suggests refactorings be composed of basic steps called micro-refactorings. Simpler transformations are easier to read, write and to verify, but on the other hand, decomposition of extensive refactorings to simple steps requires experience and considerable effort.

Strategic term rewriting. Context-free conditional rewrite rules and functional strategies [9] are widely used to implement program transformations and structured data transformations in general. Furthermore, Bravenboer et al. show that by adding dynamically defined rewrite rules into the system [8], context-dependent transformations [56] are also definable.

Indeed, traversal programming is an expressive and exciting paradigm, but as Lämmel et al. point out in their comprehensive study [46], error-free use of strategy combinators requires expertise, not mentioning the difficulties of formal verification (the termination property of a complex strategy alone is a considerably difficult problem). The paper characterises typical mistakes in strategic programming, and one of their findings is that errors mostly stem from mixing up selection of terms of interest (their type and pattern), keeping track of the origin of data, checking side conditions and doing actual transformation. Separation of these concerns is addressed in our approach.

Graph rewriting. Semantic program graphs capture the binding structure, the data and control flow relations in the program, while they may also depict properties of specific program symbols. It is apparent that semantics-aware, verifiable transformations can be specified with graph rewriting [50] as well, but the graphical descriptions of graph rewrite rules are relatively complex compared to concrete syntax patterns. In addition, matching a graph pattern to a semantic program graph is computationally more complex than matching a first-order term pattern to a term. Since the graphical format of rules is representation-dependent and rather complicated, this system is less likely to be used by users to define their own refactorings. Some systems use a graph model, but express the context-sensitive rewrite rules with a special textual representation, e.g. Padioleau et al. use a transformation language [58] incorporating semantic conditions into the textual patterns. We follow a similar route, but with significantly different formats for patterns and conditions.

Refactoring languages. Designing domain specific languages for refactoring programming is 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.

Leitão [41] gives an executable, rewrite-based refactoring language with expressive patterns, Verbaere et al. [75] propose a compact, representation-level formalism for executable definitions. These formalisms are expressive and language-independent, but at this generality they cannot support correctness checks for refactoring definitions. For Erlang, Li and Thompson [43] define an API for describing prime refactorings and a feature-rich language for interaction-aware composition, but formal verification is not addressed in their work either.

Verifiable refactoring definitions. For the object-oriented paradigm, Schaefer and de Moor introduce a system [63] in which they reason about semi-formal definitions of a set of basic refactorings. The idea of using locks and language extensions instead of preconditions is exciting and promising, but the expressiveness is limited due to the lack of custom rewritings, and full semantics-preservation cannot be expressed and proved in the system. Roberts [62] applies a different definition style, with an emphasis on the side-conditions and proper composition of the base refactorings. However, neither of them provides formally verified or executable definitions.

There are some results [66] in defining provably correct refactorings for simple languages, some mechanised proofs even for real-world refactorings [13] are available, but none of these allow for defining custom transformations and provide automatic verification for those. [23] 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 an average programmer to specify their own refactoring.

4.2 From basic to advanced term rewriting

Term rewriting serves as a great foundation for any program transformation formalism, because it can define complex changes in the program structure in a declarative way. It hides the complexity of pattern matching, variable context, replacement construction and condition checks. We build our refactoring language on the basics of term rewriting and strategies thereon. In order to help the reader understand the importance of term rewriting, as well as to prepare the presentation of the advanced features we added, we overview the basics of generic term rewriting and strategies in this section. We also point out the aspects where we improved on the current methods.

4.2.1 Term rewriting systems in general

A term rewriting system consists of an alphabet and a set of rules that can be applied to terms phrased over the alphabet. Terms are built from variables and constants using function symbols given in the alphabet. Rules consist of a matching and a replacement pattern, and may be guarded by conditions making statements about the equivalence of terms. In the scope of program transformations, terms are programs or program patterns expressed in abstract syntax.

Formally, a term rewrite system is an $\mathcal{R} = (\Sigma, R)$ where Σ is the alphabet and R is the set of rewrite rules. Here Σ defines both variables and function symbols of arbitrary arities, the symbols terms are built with. The rewrite rules in R are given in the form $l \rightarrow r$, where l cannot be a single a variable, and variables of r must be variables of l as well. Conditional rules are given in form $l \rightarrow r$ if c. The condition states equivalence (or rather equality) between two terms involving the variables present in l.

The semantics of the rewriting system defines whether a term can be rewritten to another term ($t \Rightarrow t'$). The direct rewriting relation above all terms is defined with the simple match-and-apply principle: if there exists a substitution θ mapping variables to terms such that t is a θ-instance of the term l, then t can be rewritten to t', a θ-instance of the term r.

$$\text{if } (l \rightarrow r) \in R \wedge \exists\theta : \theta(l) = t \wedge \theta(r) = t' \text{ then } t \Rightarrow t'$$

The reflexive-transitive closure on this defines the indirect rewriting relation.

Generic rewrite systems can be employed to define transformations, but they are usually used as reduction systems. In reduction systems, the goal is to obtain a normal form of the original term by exhaustive non-deterministic applications of rewrite rules. The choice of which rule to apply is made non-deterministically from amongst the applicable rules; similarly, the choice of which subterm to apply a rule to is non-deterministic. That is, in some sense, generic term rewrite systems are 'uncontrolled'.

There are two basic, important properties of rewrite systems:

- **Termination:** whether the system normalises any term without divergence, i.e. there are no infinite rewrite sequences. In the literature, lots of different limitations on the form of the rules have been considered, to provide termination guarantees. In strategic term rewriting, rules are not applied exhaustively, but there is a recursion operator, so termination is not a straightforward property in that context either.

- **Confluence:** whether the normal form is unique for any term, even for non-deterministic reduction paths. Although this property is of great importance in general-purpose term rewriting systems used for term reduction, we will not consider it, because our strategic rewrite system will be fully deterministic and does not involve exhaustive rewriting.

4.2.2 Strategic term rewriting

Term rewriting was identified as a great tool for describing different types of transformation and normalisation problems, including transformations on programs represented by means of terms. It involves exhaustively applying rules to subterms until no more rules apply, where selection of subterms to be rewritten can be non-deterministic, or may be according to some order (e.g. the innermost strategy applies rules automatically throughout a term from inner to outer terms, starting with the leaves).

The lack of explicit control of rule applications (traversal strategies) makes the system simple, there is no need to define traversals over the syntax tree, the rules express basic transformation steps and the exhaustive rewriting takes care of applying it everywhere. However, the complete normalization approach of rewriting turns out not to be adequate for advanced program transformation, because rewrite systems for programming languages will often be non-terminating and/or non-confluent. Also, in general, it is not desirable to apply all rules at the same time or to apply all rules under all circumstances.

Rather than using the rules for non-deterministic normalisation, strategic term rewriting writes simple programs (composed of strategies) that apply the rewrite rules according to a specific control. With these, one can write transformation programs that, amongst others, can implement program refactoring. Control of rewrite rule applications could be given in different paradigms; we originate our system in System S [76] and Stratego, which introduce a simple imperative language for controlling rewrite rule execution. The programs written in this system define an ordinary term rewrite system via a big-step operational semantics.

Basic strategies

Basic strategies can be regarded as control statements in a simple traversal programming language. Strategy programs are built from conditional rewrite rules and other strategies. We briefly enumerate the basic strategies, but for more details and formal semantics definitions, we refer to the paper [76] on a core language (System S) for rewriting.

There are two main sorts of strategies: for control and for traversal. For control, simple sequential combinators are introduced that implement *sequencing*, *branching (non-deterministic choice and left-choice)* and *iteration (fixed-point operator).* The semantics of these are partly based on the following property: if a conditional rewrite rule fails to match or its condition evaluates to false, the entire application of the rewrite rule fails. All control statements are failure-aware. Traversal strategies help in selecting the terms of interest for a given rewriting or strategy. They take a strategy and apply it to the subterms selected for traversal. There are basic strategies for applying a rule or strategy on the *i-th subterm of the current term, for congruence and for applying the strategy on one, some or all subterms* of the term of interest.

The basic strategies can be combined into complex strategies implementing rewrite and transformation programs on structured data. Strategic rewriting greatly improves on weaknesses of generic term rewriting, but it is still not the best formalism for defining refactoring.

In strategic rewriting, analysis and transformation concerns are not separated, queries and transformations get mixed up, which complicates the definition of complex transformations. It is also not trivial how to encode refactoring preconditions and semantic dependencies. Furthermore, in strategic rewriting, unless using dynamic rules [8], rewrite rules are context-insensitive, which makes it extremely difficult to implement transformations that have to maintain context-sensitive properties. This latter is typical in refactoring, so in our refactoring language we need to improve on the capabilities of strategic term rewriting.

Custom strategies and the lack thereof

We not only improve on the strategies of System S and Stratego, but at the same time, we restrict the strategies available in these systems. Our limitations primarily support verifiability of transformations.

Basic strategies in System S are simple, because more complex, custom strategies can be defined in terms of combinations of simple ones. For example, bottom-up traversal is expressed as $\mathit{bottomup}(s) = \mu x(\mathit{all}(s); s)$ (applying the strategy to all subterms recursively, and then to the current term), which is composed of the *all* strategy, sequential composition and fixed-point iteration. With this approach, all required strategies can be composed from a set of primitive strategies, so the strategy set is extensible.

In contrast, in our refactoring language we restrict the strategies available for combining rewrite rules, and strategies cannot be combined into other strategies. As a compensation, the traversal strategies we provide are way more expressive due to the more fine-grained program model that uses per-node references and the semantic function and predicate set. While in System S, delegation of a strategy is only possible to the subterms of the current term, in our method the node of interest can be changed to any object in the entire model by using semantic queries. In our experience, the set of pre-defined traversals and combinators we provide is sufficient for defining complex refactorings in the system, yet they remain verifiable.

4.2.3 Context-sensitive rewriting

Rewriting strategies provide control over the application of transformation rules, thus addressing the problems of confluence and termination of rewrite systems. However, another problem of pure rewriting is the context-free nature of rewrite rules. A rule has access only to the term it is transforming, but transformation problems are often context-sensitive.

For example, when inlining a function at a call site, the call is replaced by the body of the function in which the actual parameters have been substituted for the formal parameters. This requires that the formal parameters and the body of the function are known at the call site, but these are only available higher-up in the syntax tree. There are many similar problems in program transformation, including bound variable renaming, type checking, data flow transformations such as constant propagation, common subexpression elimination, and dead code elimination. Although the basic transformations in all these applications can be expressed by means of rewrite rules, these require contextual information. In this section, we demonstrate the difference between the approach of System S and our method to context-sensitive rewriting.

In strategic rewriting, context-sensitive rewriting can be achieved by the extension of rewriting strategies with scoped dynamic rewrite rules. Dynamic rules are otherwise normal rewrite rules that are defined at run-time and that inherit information from their definition context.

Example 4.2.1 (Context-sensitive rewriting with dynamic rules)**.** As an example, consider the following strategy definition [8] as part of an inlining transformation.

```
DefineUnfoldCall =
  ?|[ function f(x) = e1 ]|
  ; rules(
      UnfoldCall : |[ f(e2 ) ]| -> |[ let var x := e2 in e1 end ]|
    )
```

There above rule applied to a function definition defines a new (dynamic) rule that can transform the function call accordingly. Note that there are local variables in *UnfoldCall* that are bound outside the rule, they are not free in the rule. *UnfoldCall* in its definition is context-sensitive, because of those variables coming from the context.

Parametrising the rule. Let us rephrase the dynamic rule definition, making the dependency explicit. We note that the following rules would be invalid in Stratego, we only use them as explanation as to how it relates to our approach.

```
UnfoldCall(f,x,e1) : |[ f(e2 ) ]| -> |[ let var x := e2 in e1 end ]|
```

As the rephrased rule shows it clearly, the unfold rewriting depends on the function name, its arguments and its body. What if we could reach out to the context to gather this context information without ever visiting it in a traversal? The semantic graph model used in RefactorErl allows for such a query. Let us rephrase the previous rewriting such that we make the query explicit (informally, though):

```
UnfoldCall :
  query (f,x,e1) from the definition of the transformed call (context)
  then |[ f(e2 ) ]| -> |[ let var x := e2 in e1 end ]|
```

In our approach. Now it is apparent where the query and the transformation should happen. The rule needs to make sure that the queried information can be accessed in the rewrite rule. In our method, we do this by combining two rewrite rules, where the second rule can refer to the variables bound by the first one. Furthermore, thanks to the reference-based representation, we can easily reach the definition of the called function via a semantic graph query and get its graph node reference, so that we can use it in further steps. In our approach, unlike in System S, it is not the function definition that induces the definition of the transformation, the transformation exists independent of whether there are any function definitions or calls.

Supposed that the target node (the term of interest) of the refactoring is the function call (as opposed to the definition), we can phrase the unfold transformation in our language the following way. This combined (extensive) transformation reaches out to the definition, does matching to bind variables, and then uses the bound variables to change the call site.

```
REFACTORING unfold()
ON referred_function(THIS)
  F(Args..) -> Exprs..
THEN ON THIS
  F(ActArgs..)
  --------------------------------------
  (fun(Args..) -> Exprs.. end)(ActArgs..)
```

Such so-called extensive transformations are better expressed via refactoring schemes. We will come to the detailed presentation of the concept of schemes later in this chapter, in Section 4.3, but let us give give an example of rephrasing the above extensive transformation with the “function refactoring” scheme:

```
FUNCTION REFACTORING unfold()
ON DEFINITION
  F(Args..) -> Exprs..
  -------------------
  F(Args..) -> Exprs..
ON REFERENCE
THEN ON THIS
  F(ActArgs..)
  --------------------------------------
  (fun(Args..) -> Exprs.. end)(ActArgs..)
```

Note that the program model along with the semantic queries makes our simple set of strategies very powerful. Application of the rewrite rule can be delegated to nodes defined by the context variables or by context-sensitive queries. In the following sections, we give detailed explanation on the limited set of strategies available in our method, and as to how they are so expressive just by using a different program model.

4.3 Rewriting-based refactoring of semantic program graphs

Achieving executability and verifiability at the same time is not easy, as it is apparent from the related work. Some refactoring formalisms are too high-level: they are almost informal, hardly expressible in terms of low-level rewriting steps, although provide good basis for reasonably complex proofs. On the other hand, term rewriting based formalisms are executable, but they defines complex, non-deterministic programs on rewrite rules, hard to verify even for termination, not to mention functional correctness. We aimed at designing a formalism inheriting the best of both worlds: by building on the features of the semantic program graph representation and by incorporating some fundamental results from term rewriting research, we designed a novel refactoring language. We developed refactoring language abstractions with the following design goals in mind. The refactoring language shall be:

- *Executable*: Definitions are not only specifications, but implementations as well, meaning that they precisely determine an algorithm transforming a syntax tree.
- *Verifiable*: Definitions are able to be verified for correctness, i.e. it can be formally proven that the transformation preserves the semantics of well-formed programs.
- *Applicable*: The expressive power enables for defining complex transformations of code, not only toy examples. The language allows for defining a wide range of real-world refactoring steps.
- *Intuitive*: Only language-level concepts are used. Writing programs in the refactoring language does not require familiarity with term rewriting, static analysis or the program representation.

We fulfilled these goals by carefully developing language features of abstraction levels that express transformations independent of the underlying program model, are concrete and detailed enough to be interpreted, and are abstract enough to be handled by formal methods.

Strategic term rewriting rethought

Our language in its roots inherits a lot from strategic term rewriting, but makes it more specific, and also more general, in various aspects. Our definitions are much less representation-dependent, more liberal in term patterns, but restrict conditions to be composed of a pre-defined set of high-level semantic predicates. Also, we employ a limited set of expressive strategies.

In our solution, concerns are clearly separated: semantic analysis, semantic conditions, target node selection and transformations are given in different segments of the definition.

The description is independent of the program model used, which makes it easier to compose specifications. Rewriting rules in our system are given with rich set of patterns using concrete syntax, while conditions are phrased in terms of statements on language-level semantic properties. Complex, extensive transformations are ensured to make complete and consistent changes by using semantic dependency driven compositions of rewritings.

Interestingly enough, our restricted set of conditions and schemes make specifications semi-automatically verifiable, yet they can be translated back to concepts that can be interpreted in the transformation system. Also, the developed language is expressive enough to accommodate refactorings ranging from renaming or lifting variables to inlining or generalisation of function definitions.

Remark (Trusted components.). We emphasise that verifiability in this context applies to the refactoring logic, not its implementation. The various elements of the refactoring system, including the syntactic and semantic analysis, pattern matching and the graph transformation library, are trusted components. We do the verification in an abstract model, not the actual system. Nevertheless, if the refactoring engine is properly implemented, the changes described with the refactoring specifications in our language are guaranteed to preserve program semantics. Verification of the entire refactoring system is definitely a challenging future work.

4.3.1 Refactoring-oriented programming in a nutshell

A programming paradigm is characterized by a set of features, and the approach it uses to solve programming problems. The language we designed has some features that are not common (at least together), so we believe the language suggests a new paradigm of programming refactoring program transformations. The main characteristics of the refactoring-oriented programming paradigm are the following:

- The goal of programs is to specify semantics-preserving program transformations.
- The state space (or problem space) of the program is a program model, preferably is a semantic program graph. Both syntactic and semantic entities of the program are modelled as syntactic and semantic objects in the model, they have type-specific attributes and are accessed and modified through references.
- Refactorings are defined in terms of refactoring functions. Refactoring functions are executed on syntactic or semantic objects (terms if interest) and are expected to modify the program model in a consistent way.
- Refactoring functions can be prime, composed of rewrite steps, or can be composite, combining prime refactoring functions.
- Prime refactoring functions are preferably derived from verified refactoring schemes.
- Refactoring programmers are advised to decompose refactoring to the smallest steps possible, into so-called micro-refactorings.

The language inherited many of its features from already existing paradigms:

- *Strategic term rewriting*: The language carries the features of term rewriting in its very basics. The most fundamental building block for refactoring is context-dependent term rewriting. Nonetheless, we generalised and at the same time restricted the original definition of term rewrite rules: the matching and replacement patterns are more general in our case, the conditions are more restrictive. In addition, we made the patterns and the condition context-sensitive, such that the matching pattern and the condition can refer to variables of the rule context, as well as conditions can refer to context-sensitive properties of the term.

  Strategies took significant influence on the language, too. The way we combine rewrite rules into sequential programs has its origins in System S [76], although our formalism did not inherit the fixed-point combinator, and we added a number of combinators specifically designed for our graph-based representation supporting node references. Compared to strategic term rewriting, our rules are executed on a globally accessible term with a target being a reference to a subterm, not on the subterm. Traditionally, rewriting is pure, it takes a term and returns another term, in our case, the rewriting takes a reference and has a side-effect of changing the global term.

- *Functional influences*: Variables in rewritings are bound by pattern matching, and in prime refactoring functions (composed of rewrite rules) variables are always single assignment, that is, once bound to a value, they cannot be modified. Conditions of rewrite rules are pure, they are free of any side effects. Semantic functions and predicates can only traverse the tree, but cannot change the variables or the model. Changes to the syntax tree are completely enclosed in rewriting rules, there are no other ways to change the program representation. Multiple refactoring functions are composed by monadic sequencing, where the monad is over the program representation. We actually use a do-notation for composite refactoring functions, where each function call in the sequence is changing the global program representation.

- *Object-oriented influences*: As mentioned already, the different aspects of refactoring execution (analysis, target lookup and actual rewriting) have been separated in our method. As a consequence, our refactoring functions are meaningless without a target to be executed on. Similarly to methods executed on objects, refactoring functions are executed on nodes (subtrees) of the program graph. The target node is a hidden argument to the refactoring function, and is stored in a local variable called `THIS`. Furthermore, when calling a refactoring function, we use the well-known `obj.fun(args)` notation. In composite refactoring, variables are bound by assignment, and are mutable, in order to simplify the function definition.

4.3.2 Refactoring taxonomy

Before discussing the low-level abstractions we use in our refactoring language, we overview some high-level concepts regarding the pragmatics of the language. This involves clarifying the distinction between prime and composite refactoring, as well as the difference between local and extensive transformations. After understanding the taxonomy of refactoring, we can dive deep into the features that make up these high-level concepts.

In refactoring programming, the programmer is encouraged to divide (decompose) the refactoring to the smallest steps possible. These non-decomposable steps are called prime, or micro refactorings, they cannot be divided into smaller steps. Prime refactorings are always standalone, semantics-preserving transformation definitions. Some of them can be expressed with a single rewrite rule, while others can only be defined as a combination of multiple rewrite rules. Refactorings of the former kind define shorter, local changes, while the steps of the latter kind are called extensive.

In the related work of refactoring languages, prime refactorings are mostly considered to be already defined on a lower level (e.g. with an API, outside the language) and only the composition is realised as a domain specific language. In our system, refactorings are formally defined to the smallest step, so that the entire transformation logic can be verified for correctness.

Extensive transformations. There are prime refactorings that cannot (practically) be expressed with a single rewrite rule. This is the case when the refactoring involves changes at multiple locations in the program, and the connection among these program elements is purely semantic. Since there are dependencies among program objects, sometimes they can only be changed if all the dependent elements are changed at the same time, accordingly. We say that such changes are extensive.

Extensive transformations change interdependent parts of the program, while the changes have to maintain consistency. For example, if we rename a function at its definition, we need to change the name at all the reference sites as well, including directives, calls and other mentions. The connection between the elements to be changed is the semantic entity (the function in this case), the locations to be modified are determined by semantic relations such as "defines" and "calls". Also, this example demonstrates the typical scheme of extensive changes: there are some steps that make a twist in the semantics (changing a function name), which are then compensated by a series of additional changes (correcting the name at the call sites).

In order to simplify the definition and the verification of extensive transformations, we introduce refactoring schemes that capture the general patterns underlying similar refactorings. These schemes can be instantiated with one or more conditional rewrite rules, and expand to refactoring transformations provided that the rewrite rules meet some constraints. We elaborate on schemes in Section 4.3.5.

Composite functions. Refactoring functions can be composed into compound refactoring functions by using sequential composition and iteration. Such transformations are inherently correct, since they only execute semantics-preserving steps in some order to some program elements of interest.

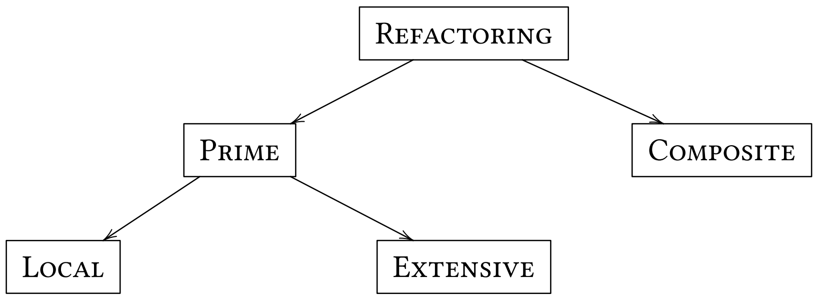


Figure 4.1: Taxonomy of refactoring

Figure 4.1 sums up the categories of refactoring transformations. In the following subsections, we address the various features of the language one by one, and explain how they can be interpreted in a refactoring engine and how they are verified in a formal system. This overview is bottom-up: we start from the smallest building blocks (such as term rewrite rules), and move towards the more complex elements (refactoring and selector functions).

4.3.3 Rewrite rules

The fundamental and most essential building block of our formalism is conditional context-sensitive term rewriting. The format is very similar to that of seen in ordinary conditional term rewriting:

$$l \rightarrow r \quad \text{when} \quad c$$

We use the following syntax for scripting rewrite rules in our refactoring language. The left pattern (matching pattern) and right pattern (replacement pattern) are separated by a line composed of dashes, while the condition follows after the `WHEN` keyword. We will cover more details on how patterns and conditions are composed.

```
  <matching pattern>
  --------------------
  <replacement pattern>
WHEN
  <condition>
```

The semantics of these rewrite rules is the same as that of the classic rules, except that the unifying substitution θ may map lists of terms to variables (as opposed to single terms), and in addition, a variable-to-value context γ has to be taken into consideration.

Remark (Term rewriting on graphs). Rewrite rules of the above form define term or tree rewriting, not graph rewriting. On the other hand, our program model in the refactoring system is a semantic program graph. We remark here that the semantic program graph includes the entire syntax tree, therefore we can do the match and construct logic on the syntactic layer of the graph, while the semantic layer of the graph is recalculated by the static analysis engine to restore the consistency of the graph.

Patterns

Patterns are first-order terms: generalised syntactic terms involving variables that can match arbitrarily compound subterms. Patterns in our language are always given in concrete syntax. Variables appearing in patterns are called metavariables, as opposed to variables in the transformed language. In our current notation, metavariables are denoted by Erlang variables, whilst literal variables are matched by using a special semantic predicate. Consequently, metavariables follow the naming rules of Erlang variables and start with a capital letter.

Example 4.3.1. Consider the following unconditional rewrite rule (the condition is the *true* literal), which swaps any two expressions connected by an addition operator.

```
X + Y
-----
Y + X
```

Applying the rule on $[\![1 * 2 + 3]\!]$ and supposing an empty environment (both X and Y are free variables), the matching θ would be $[X \to 1 * 2, Y \to 3]$ and the resulting expression would be $[\![3 + 1 * 2]\!]$. However, similarly applying the rule on the same expression, but in a context $\gamma = [X \to 3]$, the matching would fail, since unification of $1 * 2$ and 3 is not possible.

Remark (Equivalence in unification). In our matching algorithm, unification uses equivalence checking rather than strong equality or join-ability. The algorithm allows for type conversion between values and program entities, such that a semantic function object is treated equivalent to its signature, or a syntactic integer literal is equivalent to its value. Equality of nodes (subterms) is defined recursively as structural equivalence.

Remark (Implicit conversion in replacement). Automatic conversion between literal nodes and literal values is not only done in matching and unification, but also in building replacement terms. For example, when a metavariable X holds the value 1 of type *integer*, in a replacement term containing X, it will be converted to a syntactic expression object, a literal of value 1.

Example 4.3.2 (Non-linear patterns)**.** For the sake of increasing the expressiveness of patterns, the language allows non-linear patterns in rewrite rules. For example, the following rewrite rule has both its matching pattern and replacement pattern non-linear.

```
  A, A, B
  -------
  A, B, B
```

Nonetheless, at some aspects, this causes unreasonable complexity in the implementation, and the verification part does not support it fully either. It is recommended to use equality checks and matching conditions instead of relying on non-linear patterns. Rephrasing the above rewrite rule to only use linear patterns and additional conditions results the following rule:

```
  A, B, C
  -------
  A, D, C
WHEN
  A = B AND D = C
```

List metavariables

We syntactically and semantically distinguish variables that hold a list of values or terms from those that store a single value or term. While ordinary metavariables match and record exactly one syntactic subterm (subtree) or value, list metavariables (denoted by postfixing the variable name by two dots) can match and store zero, one or more consecutive, sibling subterms, i.e. a list of values. Otherwise, they are used the same way as single-value variables, can appear in patterns and conditions.

By distinguishing list metavariables, our language can support associative-commutative matching (although verification support is not complete for this yet): when multiple list metavariables are matching a list of nodes, pattern matching results all the valid combinations.

Example 4.3.3. Matching $[A.., B, C..]$ against $[1, 2, 3]$ would produce the following results (tupled values of A, B, and C respectively): $[([], 1, [2, 3]), ([1], 2, [3]), ([1, 2], 3, [])]$. When the result of the matching contains multiple valid results, it is the responsibility of the condition to invalidate all but one. For instance, adding the condition $\mathit{length}(A..) = 1$ would deterministically select the second option. Similarly, matching the pattern $[A.., 2, C..]$ against the above ground term would determine the very same single solution.

Let us mention a practical use case to this feature. In automatic parallelisation, we used such patterns to match on formal arguments of recursive function definitions processing lists, where it was not known at which position the processed list is present in the argument list. The following pattern provided solution, by generic matching and a strong enough condition on the variable L. Such a pattern let us localise the processed list in the argument list and treat specially in the rewriting.

```
Fun(Args.., L, Rest..) WHEN is_list(L) AND recursively_consumed(L)
```

Example 4.3.4 (Metavariables and context query)**.** The following rewrite rule matches simple (module-local) function applications and turns them into module-qualified (external) calls, making it explicit which module the called function belongs to. Since we match the arguments with a list metavariable, regardless of how many arguments the invoked function takes (zero or more), the expressions of actual parameters are simply reused in the new call.

```
  Fun(Args..)
  ---------------
  Mod:Fun(Args..)
WHEN
  atom(Fun) AND Mod = module(THIS)
```

Semantic functions and predicates

An important concept in our approach is providing a language-specific, pre-defined set of semantic functions and semantic predicates. Semantic predicates provide the predicate set available in rewrite rule conditions, they are applied to node references and return true or false. Semantic functions are used both to lookup node references and to query semantic data (e.g. find a function by its name, or query the name of a function from its object).

These are pre-defined and language-specific on purpose. Side-conditions of refactorings are usually phrased like *"F is an exported function"*, *"expression A depends on expression B"* or *"expression E is pure"*. It is apparent that an easily readable formalism has to provide the same abstraction level as the human has when considering the conditions of a transformation. We defined a set of such functions for Erlang, which we believe are sufficient to express a reasonably large variety of transformations.

Example 4.3.5 (Semantic predicate syntax and semantics)**.** Semantic predicates are incorporated both in execution and verification of rewrite rules. To explain how, let us consider an example property of an expression being pure is defined by the semantic predicate *pure/0*. Applying this to a syntactic or semantic object is returns with the fact whether the object, when executed, can have any side-effects.

From the operational semantics point of view, there is a static analysis algorithm (or an inductive definition) behind telling whether an expression is free of side-effects. This analysis checks if the expression affects the state, raises exceptions or refers to any impure internals (e.g. reading or writing IO). The predicate evaluates to *true*, if the expression does not have any side-effects on the global state, while evaluates to *false*, if it may have side-effects [1].

[1]In the refactoring engine, there is differentiation between expressions that *may have side-effects* and those that *definitely have side-effects.* More on this in our paper [4] written on automatic parallelisation of Erlang programs.

From the verification point of view, axiomatic semantics of the predicate is more important. For purity, the following statements are added to the set of valid rules (formulas) in the verification system:

$$\begin{array}{rcll} \texttt{E} & \Longleftrightarrow & \texttt{(fun() -> E end)()} & \text{when} \quad \mathit{pure}(E) \\ \texttt{E, Es..} & \Longleftrightarrow & \texttt{Es.., E} & \text{when} \quad \mathit{pure}(E) \end{array}$$

This expresses that if an expression is pure, it does not matter where in the evaluation order it is visited, and it can be unwrapped if it is enclosed by a function abstraction. When refactoring definitions refer to purity in the condition, the verification system takes the above equalities into account and tries to use them in the correctness proof.

Amongst others, there are semantic functions for querying properties of semantic entities such as modules, functions or variables, while predicates tell whether particular relationships exist between program units. Semantic functions and predicates are built-in and have a well-defined semantics; user-defined functions cannot be added, the system is not extensible in this sense.

Conditions

Rule conditions control whether a rewrite rule can be applied. In the refactoring terminology, conditions specify the so-called side-conditions of transformations, that are required to be met for the transformation to be semantics-preserving.

Conditions are first order logic formulas built upon semantic functions and predicates. Formulas are applications of semantic predicates, or equivalence checks on values of expressions; they are composed by negation (*NOT*), conjunction (*AND*) and disjunction (*OR*). Expressions include constants, metavariables and applications of semantic functions. Formulas are evaluated left-to-right, call-by-value.

Example 4.3.6. Consider the following simple rewriting rule that expresses the expression unwrapping in a different form. It matches an argument list to the function closure, but only allows the rewriting if it is empty. This means that the rule can be applied to anonymous functions having a non-empty argument list, the matching succeeds, and the condition will make the rule fail.

```
  fun(Args..) -> E end
  --------------------
                 E
WHEN
  pure(E) AND length(Args..) = 0
```

The verification of this rewrite rule can be performed based on the language semantics and the axioms presented above for the *pure* predicate.

Proposition 4.3.1 (Verifiability of local refactoring)**.** *Local refactorings defined with a single conditional rewrite rule expressed in the above formalism are automatically verifiable for semantics-preservation.*

Proof. It is detailed in [29] that we can derive the correctness of a rewrite rule to equivalence of two expression patterns under a given condition. Furthermore, we can reduce the previous equivalence property of the two expression patterns to a correctness property of an aggregated program constructed by the two expression patterns. Then we can apply a language-independent, general-purpose proof system to automatically check the validity of the property. The semantic predicates and functions available in conditions are axiomatised in the proof system. □

Remark (Equalities in conditions). In ordinary term rewriting rules, equality checks in conditions test whether the two sides are the same, or their normal forms are equal. This check relies on the same relation that is defined by the rule, so checking the condition may result in infinite recursion. In our language, formulas cannot refer to the rewriting relation, equality checks are performed by structural or value-based comparison.

Matching or binding conditions. Advanced conditions may have a kind of side-effect: if the left hand side of a condition (equality check) is a free variable, the value of the right hand side is bound to the variable. Note that metavariables bound this way can be used in the replacement pattern to contribute to the new subtree, so this way some context-dependent information can be gathered for the transformation in the condition. The following example demonstrates a use case for matching conditions.

Example 4.3.7 (Binding condition)**.** The following refactoring rewrites an Erlang list comprehension into an application of the *map* higher-order function, whereas the generated list and the head function are extracted into auxiliary variables (`List` and `Fun`).

```
  [ Head || GeneratorsFilters.. ]
  ------------------------------------------
  List = [{ Vars..} || GeneratorsFilters..],
  Fun = fun({Vars..}) -> Head end,
  lists:map(Fun , List)
WHEN
  Vars.. = intersect(bound_vars(GeneratorsFilters..), vars(Head)))
  AND fresh(List)
  AND fresh(Fun)
```

Note that *Head* matches arbitrarily complex expressions, while *Vars..* captures all variables that are bound by the comprehension generators and are referred to in the comprehension head. The lists of variables returned by the semantic functions *vars* and *bound_vars* are intersected according to set intersection; the ordering in the final result is undefined — and irrelevant in this particular case.

Via the condition stating that the variables *List* and *Fun* are fresh names, the transformation guarantees that the newly introduced variable names are not bound in the scope of the transformation. Note that this condition is context-sensitive, since its validity is dependent on the context of the term of interest. It is a special form of binding condition, which binds the metavariable to a randomly generated variable name having the metavariable name as prefix.

4.3.4 Combinators and modifiers

Since complex data and control dependencies are present among the various elements of the program, some transformations can only be complete and correct if all the dependencies are handled properly when the origin of the dependency changes. This requires combination and control of rewrite rule applications, via methods similar to strategies. We define the so-called extensive transformations by combining conditional rewrite rules via combinators and modifiers, which are specialised, reference-based strategies. The following strategy-like constructs are available in our methodology.

- *sequencing*: The keyword *THEN* makes sequential composition of two rewrite rules. The semantics is similar to that of in System S: *A THEN B* executes A first, and if it succeeds, executes B as well.
- *left-choice*: The keyword *OR* creates the left-biased composition of rewritings. In semantics, *A OR B* executes A, and proceeds to B only if A has failed for some reason.

We did not include non-deterministic choice and general, unbounded recursion (fixed-point combinator) into our strategy set, because these bring the main complexity into the verification process, yet we can express most transformations without them. The former one is not a desired feature of our language, while the latter is left out due to its very generic nature. Although without the fixed-point operator our language is not Turing-complete, the bounded recursion implemented with modifiers and selectors provides enough flexibility for refactoring definition.

Modifiers can be regarded as advanced traversal strategies. With combinators, one can compose rewritings, while with modifiers, we control the target of the rewriting. Without modifiers, rewrite rules apply on the target of the refactoring function, but with modifiers, one can change the node/term of interest. Modifiers evaluate expressions that determine node(s) on which the rule is applied:

- *one-level traversal*: The modifier *ON* takes an expression, evaluates it (the result should be a node reference or a list of node references), and sets the target of the rule to the result. If the expression evaluates to a single node, the rewrite rule is applied on the subtree determined by the node. If the result of the expression is a list, the rewrite rule is executed on each element of the list; on empty lists, it takes no effect.

- *multi-level traversal*: The modifier *IN* is very similar to the *ON* modifier, but the rule is applied not only on the result of the expression, but on all the nodes within its subtree, recursively. The semantics is terminating, as the subtree is traversed and the target nodes are collected before applying the rewrite rule. That is, nodes synthesised by the application of the rule are not added to the target set.

Remark. The *IN* modifier can be suffixed by the keywords *BOTTOMUP* and *TOPDOWN*, in order to control the order in which the subtree is visited. The default is top-down.

Both *IN* and *ON* may be suffixed by the keywords *ALL* and *ANY*, in order to control failure handling. When the latter is used, the application of the rewriting rule accepts failing cases (but requires at least one succeeding case), while the former one requires the rewriting to succeed on all targets (this is the default behaviour).

Remark. There are benefits and drawbacks lying in the semantics of *IN*. Although the pre-collection of target nodes is simple and prevents divergence, it may cause null reference exceptions when it reaches a subtree that has been changed in a preceding iteration. In such cases, the traversal ignores the exceptional case and proceeds with a warning.

Example 4.3.8. The following example demonstrates simplified variable inlining by using combinators and modifiers. This extensive rewriting can be applied to expression lists starting with a simple variable binding, and after removing the binding (match expression), it continues by traversing the rest of the expression list to propagate the right hand side of the assignment to the references of the removed variable.

```
  X = Y, E..
  ----------
  E..
WHEN var(X)
THEN IN E..
   X
  ---
   Y
```

Remark (Transaction-aware variants of combinators). For technical reasons, there additional combinators and modifiers in the refactoring language, which control the finalisation of transactions in the refactoring system. Semantic analysis is only executed at the end of each transaction, so that the refactoring language helps make explicit control of when the semantic layer has to be re-analysed. There is a sequencing combinator called *ALSO* which postpones semantic analysis, while there is a modifier called *SIMULTANEOUSLY*, which keeps changes made by a traversal in one single transaction.

Definition 4.3.1 (Executable refactoring specification)**.** We say that a refactoring specification is executable, if it can be given a mathematically precise meaning which defines a program-to-program transformation in terms of an algorithm or function on the program model.

Proposition 4.3.2 (Executability of prime refactoring). *Local and extensive transformations defined by context-sensitive conditional rewrite rules, combined and modified with the above discussed control and traversal strategies are executable in a refactoring system that incorporates a semantic program graph model.*

Proof. Local refactoring definitions are defined with a single conditional term rewrite rule. The target node (determining a term of interest) is always a syntactic object (determining a syntactic subtree), on which the term rewriting can be done with the ordinary semantics. Conditions are given as semantic predicates on expressions composed with semantic functions, each defined on the semantic program graph as executable, type-unifying and type-preserving queries.

Extensive refactoring definitions combine executable conditional rewrite rules. The combinators inherit their semantics from System S (essentially, sequential composition of transformations), while modifiers implement simple traversal and iteration with selectors that are composed of variables and executable semantic functions. □

Remark. It is not straightforward that semantic functions and semantic predicates are always computable and their computation terminates. In our implementation, program semantics is approximated in a finite amount of time, which is guaranteed to terminate.

Remark. The ON and IN modifiers combined with the *children* one-level syntactic traversal selector can show the behaviour of the System S compound combinators. For example, the traversal *ANY ON children(THIS)* shows similar behaviour to the strategy combinator *some* in System S, while *ANY IN children(THIS) TOPDOWN* is similar to *sometd*.

4.3.5 Refactoring schemes

Extensive code transformations can be expressed with traversal strategies, strategy combinators and complex semantic queries. However, they are basically as hard to be verified for semantics-preservation as any structured sequential program implementing a refactoring. Since our goal was to provide a tool set in which all refactorings can be semi-automatically verified, we need to restrict extensive transformations so that they become automatically verifiable.

The idea: generic semantics-driven extensive change

In a typical extensive refactoring, the modifications that have to be carried out simultaneously (connected with rewrite rule combinators) are related from the semantics point of view. The connection between dependent parts is determined by some semantic entity or relationship, such as data-flow and control-flow. As a matter of fact, this induces grouping among extensive refactoring steps: some extensive transformations change functions, while others alter data-flow or control-flow.

We exploit this idea and make generic extensive refactoring steps. The generalisation is on the actual rewriting the transformation applies on the interdependent parts of the program. The generic part of the transformation is responsible for traversal control (ensuring the changes are complete), while the specific part determines the modification to be made at the interdependent locations of code (specific parts of the transformation need to define consistent changes).

A motivating example: renaming functions

Let us illustrate the idea with the following motivating example. Suppose that we would like to write an extensive refactoring that can rename a function. Assume that the name of the function to be renamed is given in the variable *OldName*, while the new function name is given in the variable *NewName* (for the sake of simplicity, we first assume that the name alone identifies the function, but later on we also add module name and arity).

The following rewrite rules should be combined some way to rename the function both in its definition and its calls. For the function definition with the signature and the function body expressions:

```
  OldName(Args..) -> Exprs..
  --------------------------
  NewName(Args..) -> Exprs..
```

And for the function calls with the function name and the actual arguments:

```
  OldName(Args..)
  ---------------
  NewName(Args..)
```

Individually, they are transformations causing inconsistency, but properly combined, they can make an extensive, complete, consistent change in the program.

We can use the combinators discussed in the previous section to create an extensive transformation. With the combinator *THEN* we compose an extensive rewriting from the two rules, while by using *ON* we target the rules to their targets. It is not straightforward what the selectors for the modifiers and the condition for the transformation should be.

```
ON (modifier?)
  OldName(Args..) -> Exprs..
  --------------------------
  NewName(Args..) -> Exprs..
WHEN (condition?)
THEN ON (modifier?)
  OldName(Args2..)
  ----------------
  NewName(Args2..)
```

Note that in the combined version, we need to change the argument-matching metavariable in the second pattern (from *Args..* to *Args2..*), otherwise the system would try to match the formal parameters with the actual parameters by syntactic structure, which may or may not succeed, but it is not intended. We also need to add a condition that prevents name clash, which is an obvious side-condition of the renaming refactoring, and we should provide modifiers that delegate the rewriting rules to the definition and the calls of the function.

We can rely on the static semantic information available in the semantic program graph, and the semantic functions and predicates defined thereon: *function_definition* and *function_references* point from the function to its defining clauses and calls, respectively, whilst *function_exists* is a predicate that tells whether a function in a particular module with the given name and argument count exists or not.

The following definition shows the extensive refactoring transformation that renames a function. Observe that extensive refactoring is always semantics-controlled: in this case, the locations to be changed are connected by the semantic function object stored in variable *Fun*. The variable *OldName* is not necessarily a parameter of the rewriting any more, but the new name is still expected to be given in *NewName*. When wrapping this extensive step into a refactoring function, *Fun* will be an implicit parameter, while *NewName* will be an explicit parameter.

```
ON function_definition(Fun)
  OldName(Args..) -> Exprs..
  -------------------------
  NewName(Args..) -> Exprs..
WHEN NOT function_exists(module(Fun), NewName, length(Args..))
THEN ON function_references(Fun)
  OldName(Args2..)
  ----------------
  NewName(Args2..)
```

The above specification visits the definition of the function, checks if the new name is not taken yet, and if the condition is true, renames the function in its definition. Then visits all calls to the function and changes the function name to the new one. This way, the changes made in the program are consistent, because we changed the function name at all locations accordingly. Correctness of this transformation can be verified manually.

Extracting the scheme. Now observe that this extensive refactoring induces a scheme: refactoring a semantic function entity by changing the function signature in both its definition and its references. We can image a lot of different rewritings that we can apply on the definition and on the calls of the function that change the code consistently, which would result in different refactoring definitions. Indeed, this is a scheme (or skeleton) of extensive rewriting.

```
ON function_definition(THIS)
  P1 -> Exprs..
  -------------
  P2 -> Exprs..
WHEN C
THEN ON function_references(THIS)
   P1
  ----
   P2
```

As you can see, such a scheme is parametrised by a rewrite rule $P_1 \rightarrow P_2$ if C, which is applied to both the definition and the references of the function with a proper control and traversal. In fact, we can instantiate this scheme with several different rewrite rules to obtain different refactoring transformations. Rewriting the name in the signature gives us the "rename function" refactoring.

```
FUNCTION SIGNATURE REFACTORING rename(NewName)
  Name(Args..)
  ---------------
  NewName(Args..)
WHEN
  NOT function_exists(module(THIS), NewName, length(Args..))
```

Rewriting the argument list by coupling the arguments in a tuple gives another instantiation of the scheme, namely, a refactoring known as "tuple function arguments".

```
FUNCTION SIGNATURE REFACTORING tuple_function_arguments()
  Name(Args..)
  --------------
  Name({Args..})
WHEN
  NOT function_exists(module(THIS), Name, 1)
```

There is a more general variant of this scheme, which is parametrised by two separate rewrite rules that change the function definition and its references. Several schemes will be discussed in the subsequent parts of this section.

The essence of schemes

Schemes can be understood as complex strategies in traversal programming, but in fact they are much more: schemes define the format of their parameter rewrite rules and may inspect the elements of the rules in order to define the compound strategy they carry out. They are verified parametrised by arbitrary rewrite rules complying with some contracts, such that instantiated with contract-complying rewrite rules they are guaranteed to be semantics-preserving. Apparently, verifying a scheme is a difficult task, but it then provides a verified skeleton for further refactorings.

Schemes make sure that the transformation will reach out to all code locations that might be affected, and also make sure that the changes made are consistent. Intentionally, schemes hide the complexity connected to semantics-based term selection and side conditions, while at the same time they fully control the application of rewrite rules by relying on these semantic connections. Schemes are instantiated with a series of conditional term rewrite rules, which are expressed in the concrete syntax of the object language, and they may refer to pre-defined semantic functions and predicates. These latter provide access to the program representation with an interface that resembles object language level concepts, allowing anyone knowing the object language to read and write refactoring definitions.

Proposition 4.3.3. *Extensive refactorings defined by instantiation of pre-verified schemes are semi-automatically verifiable for correctness.*

Proof. The methodology of the two-phase verification of scheme-based extensive transformations is discussed in [30]. □

Schemes currently defined in the system

In order to implement some complex case studies, we defined a number of schemes we could instantiate extensive refactorings with. We decided to add a scheme even for local refactoring, which makes us define each and every prime refactoring with a scheme, either local or extensive. Most schemes are based on dependencies among program fragments. In general, they fall into one of the following categories:

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

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

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

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

The above classification of schemes intentionally uses language-independent concepts, such as data-flow and name bindings. There is already some effort put in making the entire methodology available for other paradigms and languages.

Although most schemes are described and illustrated with examples in Section 4.4 (the compound refactoring case study), in the following example we provide a detailed explanation of the forward data-flow scheme in order to help the reader get familiar with the core idea of making these steps executable and verifiable.

Example 4.3.9 (Forward data-flow refactoring scheme and instance)**.** The data-flow schemes are based on the data dependency induced by data-flow. In the forward change scheme, the dependency is followed from the origin to the references. If an expression constructing a value is changed, all the expressions into which the value flows (and therefore induces data and behavioural dependency) should be changed as well.

This skeleton is parametrised by a number of rules applied to either the construction site or a reference site of the data. That is, one of the definition rules is applied on the defining expression (the target of the refactoring), while the expressions referring to the data are transformed by one of the reference rules. In our current model, all elements on the data-flow path starting with the expression constructing the value are regarded as references. If the definition or any of the references cannot be transformed by a corresponding rule, the refactoring fails.

The syntactic skeleton is the following (there can be multiple rewrite rules for the definition and the references, but for the sake of simplicity, we present the simplified skeleton).

```
FORWARD DATAFLOW REFACTORING <name>(<arguments>)
DEFINITION
   <pattern1>
   ---------- WHEN <condition>
   <pattern2>
REFERENCE
   <pattern3>
   ----------
   <pattern4>
```

There is an important side-condition for this scheme. Refactorings created with it will fail when any of the references to be compensated have any data sources (i.e. preceding data-flow nodes) other than the originally selected refactoring target. In the expansion of the scheme, we use the predicate *single_source* that does backward data-flow reaching to determine if there is only one data origin. The metavariable *CURRENT* holds a reference to the node currently transformed by the iteration. After instantiation (expansion), we get a refactoring function composed of the patterns and conditions specified in the parameter rewrite rules.

```
REFACTORING <name> (<arguments>)
ON THIS
   <pattern1>
   ----------
   <pattern2>
WHEN <condition>
THEN ON fw_dataflow(THIS)
   <pattern3>
   ----------
   <pattern4>
WHEN single_source(CURRENT)
```

Let us see a concrete example refactoring that is defined with the forward data-flow scheme. By instantiating the scheme, we can define a transformation that can eliminate an anonymous function unnecessarily wrapping a pure expression. The definition rule extracts the value, while the reference rules take care of the applications of the anonymous function. With a similar refactoring definition, we might inline the unnamed function by referring to the body of the function in the reference rules. (The explicit mentions of variables F and G after the keyword `REFERENCE` are needed for verification purposes and target node lookup.)

```
FORWARD DATAFLOW REFACTORING fun2value()
DEFINITION
   fun() -> E end
   -------------
         E
  WHEN pure(E)

REFERENCE F
    F()
   ----
    F

REFERENCE G
   apply(G, [])
   ------------
         G
```

Execution. Let us demonstrate the effect of applying the extensive refactoring. Executing the “fun2value” refactoring on the fun expression checks if the value “apple” is side-effect free, and then it removes the unnecessary abstraction and application. Changes are carried out both in the data definition node and on its reference.

```
X = fun() -> apple end,
... ,
atom_to_list(X())
```

$$\xrightarrow{fun2value()}$$

```
X = apple,
... ,
atom_to_list(X)
```

Verification. In many cases, the definition and reference rules are inverse in some sense, which is the intuitive understanding of the verification method. The instantiation contract of the data-flow scheme is that the definition and reference rules make consistent changes (expresses as equivalence formulas composed by combining the matching and replacement patterns of the rewrite rules specified for definitions and references). For “fun2value”, the instantiation is said to be correct if following formulas are valid (these formulas provide the denotational semantics of the extensive refactoring defined by the scheme).

$$\texttt{(fun() -> } E \texttt{ end)()} \equiv E$$
$$\texttt{apply(fun() -> } E \texttt{ end, [])} \equiv E$$

These formulas express conditional equivalence between expression patterns, which can be automatically verified [30].

4.3.6 Refactoring functions

Scheme instances can be given a name and can be parametrised by variables of any type (including node references as well), resulting in so-called extensive refactoring functions. These are complete specifications, determining the algorithm that is a correct, semantics-preserving transformation step. Such functions can be combined similarly as individual rewrite rules have been combined, resulting in compound refactoring functions.

```
REFACTORING <name> (<parameters>)
  <rewriting rule(s)>
```

Refactoring functions are identified by their name and arity (number of parameters). They have an implicit parameter, the node of interest of the transformation (variable *THIS*), but may have any number of additional arguments; the return value is the node(s) changed by the refactoring.

Transformation or refactoring? It is worth clarifying that the generic language we built upon term rewriting can define non-refactoring transformations, but functions that are instances of schemes are provably correct. Therefore, functions instantiating schemes are always refactorings, not only in their name, but in their semantics, too.

Compound refactoring. Refactorings functions can be composed of sequential applications of other refactoring functions. Calls to other refactoring functions composed with the combinators and modifiers introduced in the previous sections form a compound refactoring transformation. The syntax is a bit different, but in compound refactoring functions, other functions instead of rewriting rules are executed according to some strategy. The keyword *DO* indicates that the refactoring function is compound (and therefore requires no verification).

Sequential composition is implicit, the *THEN* keyword is not needed. However, there is another control statement available in compound refactoring: the keyword *ITERATE* keeps executing the supplied refactoring function until it fails. As a consequence, compound refactoring is correct if it terminates, but it is not guaranteed to terminate (i.e. total correctness is not guaranteed).

```
REFACTORING <name> (<parameters>)
DO
  [ <var> = ] <selector>.<function>(<parameters>)
  ...
  [ <var> = ] <function>(<parameters>) [ <modifier> <selector> ]
  ...
  [ <var> = ] ITERATE <selector>.<function>(<parameters>)
```

As the above syntactic skeleton suggests, refactoring functions can be applied to nodes with two different syntaxes: an object-oriented style notation (meaning an *ON* modifier on the selector), or modifiers can be explicitly written after the call (either *ON* or *IN*). These modifiers have the same semantics as seen previously (*ON* applies the function on the term determined by the selector, while *IN* executes the function on all subterms of the term determined by the selector).

The result of a refactoring call (node references) can be stored in function-local mutable variables to be used later on as parameters or in selectors.

4.3.7 Selectors

Selectors are expressions that evaluate to node references. Syntactically, they can be either references to variables or applications of semantic and selector functions.

Selector functions. Selector functions do traversal, pattern matching and condition checking, without any modifications of the graph. This is a special language construct that provides node lookup and branching in a rather unique way.

```
SELECTOR <name> (<parameters>)
<modifier> <selector>
  <pattern> WHEN <condition>
RETURN <selector>
```

Selector functions behave very similar to refactoring functions, but they are pure, have no side-effects of modifying the program model. The target node is a hidden, implicit argument to the selector function, just like to refactoring functions, and selector functions explicitly return node reference(s) to be further processed by other functions.

Remark (Selectors instead of branching statements). As it is apparent now from this chapter, our refactoring programming language does not have if-like or switch-like branching. However, the branching control structure is present in the language: if the application of a function call depends on a condition expressible as a boolean formula, it can be put into the condition of a selector, so that if the condition is false, the selector returns an empty set of nodes and thus the functions does not get executed.

Proposition 4.3.4. *If all prime refactorings are defined as instances of schemes, the entire refactoring program is semi-automatically verifiable for correctness.*

Proof. Instances of schemes are semi-automatically verifiable, so if all prime refactorings are defined with scheme instances, all prime refactorings are semi-automatically verifiable. Compound refactorings only change the program model via calling refactoring functions (selectors cannot have effects on the semantic program graph), but the functions they may call are semi-automatically verifiable (inductive hypothesis), and if those are correct, their sequential composition and iteration (defining a series of semantic-preserving transformations) is inherently correct. If all refactoring functions are correct, the entire program is correct. □

Pros and cons of using node references

The definition of a refactoring transformation highly depends on the abstraction level and richness of the used program model: the more advanced the model is, the easier to express preconditions and transformation steps.

We decided to use the semantic graph model, and manipulate it via node references. There are some complications that stem from using strategic term rewriting in an impure way, modifying a global graph rather than implementing pure transformations on terms. Nevertheless, there are some clear advantages as well, so we enumerate some of them in the following paragraph.

On the positive side, we have efficiency and simpleness, especially compared to ordinary strategic term rewriting.

- *Memory-efficiency*: When terms are passed to rewrite steps or functions, they are passed by reference. Thus, no copy takes place when a syntactic or semantic object is passed to another function in the program, which is efficient both in time and memory. Semantic properties and related nodes can be easily accessed through the reference.

- *Easy lookup of context*: When transformations need to look out for their context in order to check their context-sensitive condition or find their target node of interest, they can use graph-wide queries to obtain the required information easily and effectively. Semantically interdependent syntactic parts are always connected in the graph, therefore relevant semantic queries only take one or two traversal steps.

- *Easy query of semantic properties*: Semantic properties are usually complex to tell, but in our refactoring engine, approximated static semantics of programs are already stored in the semantic program graph as additional nodes and edges. Semantic nodes store their properties in their object attributes. Therefore, querying semantic properties about any syntactic elements takes little effort.

Manipulating a global data via references may be effective and convenient, but need careful consideration and experience in some situations.

- *Re-matching*: Let us recall Example 4.3.1, which defined a rule that swaps arguments to an operator. We stated that applying the example rule on $[\![1 * 2 + 3]\!]$ results the matching $[X \to 1 * 2, Y \to 3]$. Actually, the matching will bind variables to node references, and not values, and the system has to work with the references afterwards. In some cases, the replacement pattern has to be re-matched on the result subtree in order to update reference identifiers in metavariables present in the replacement pattern. A typical scenario for this is when replacement pattern is not linear, and construction creates copies of nodes with new identifiers.

- *Changing context*: A node that is affected by rewriting does not necessarily disappear, but may be get moved in the tree. Consequently, metavariables holding a reference to a node, may receive different answers executing the very same query before and after a transformation. The explanation to this is that the queries on the nodes may be context-sensitive, and even if the node itself is not changed, its context may be altered.

- *Null pointers*: In complex refactoring functions, dozens of variables of different scopes are bound to various node references. Although rewriting makes sure to track changes and update the corresponding references, in some cases, references get invalid. Unfortunately, the garbage collection implemented in the refactoring engine cannot properly take into consideration the references hold by metavariables in the refactoring language interpreter.

- *Impurity*: Last but not least, impure refactoring functions are not straightforward to be run in parallel, in contrast to System S programs, which in theory are completely pure and their concurrent execution is simple due to the lack of dependencies. Nevertheless, adding dynamic rules to System S introduces dependencies and impurity; thus, real refactoring definitions are not amenable to parallelisation with dynamic rules either.

4.3.8 Notes on the formal semantics

The semantics of the refactoring language can be approached both from the operational and from the denotational perspective. Traditionally, operational semantics defines how a program in the language is executed, while denotational semantics maps the program to a so-called denotation that captures the meaning.

Operational semantics. Defining an operational semantics for the language would definitely justify that the specifications are executable. We have already addressed the question of giving a big-step semantics to the refactoring language, but it has not been published yet. This semantics would define a transition relation that mimics the operational semantics of term rewrite systems.

Nevertheless, the language already has an operational semantics determined by its implementation in Erlang, although it cannot be regarded a formal definition.

Translation semantics. It would be worth investigating whether the semantics of the refactoring programs can be expressed in terms of graph rewrite rules or programs in System S. Beyond questions, graph rewriting over an extended program model seems to be a proper domain to express refactorings, but as Stratego suffers defining complex transformations, expressing our entire query and transformation logic in System S with dynamic rewrite rules would be rather challenging.

Denotational semantics. In our current understanding, the denotational semantics of refactoring specifications in our language is a set of conditional equivalence formulas (in matching logic) that are to be valid for the transformation to be correct. All scheme-based prime refactorings should be mappable to such a formula set, local transformations to single formulas, while extensive refactorings to a set of multiple formulas.

Although this kind of verification method is already well-designed, the prototype is not stable yet. However, our proof of concept demonstrates that we can indeed transform our specifications to automatically verifiable formulas.

4.3.9 Notes on the verification method

In our methodology, prime refactorings are mapped to equivalence formulas to be verified based on the formal semantics of the object language. Local refactorings are mapped to formulas of a very similar form to the rewrite rule they define, while extensive refactorings are verified in two parts: verification of the scheme and verification of the instantiation of the scheme. The former proves that the refactoring is complete, whilst the latter guarantees that the refactoring makes consistent changes; we identified these two properties to be fulfilled at the beginning of the chapter.

The verification of refactorings is not my particular result, my colleague, Judit Kőszegi worked on developing the method of producing proofs for all sorts of refactoring. Let me briefly summarise the verification technique. In [29] we presented how to use a proof system to prove refactorings whose correctness can be expressed by the equivalence of two expression patterns under a given condition. We reduced the equivalence property of the two expression patterns to a correctness property of an aggregated program constructed by the two expression patterns (according to [12]), then we applied the language-independent, general-purpose proof system to automatically check the validity of our property.

According to our terminology, refactoring correctness is defined with respect to a formal semantics of the object language and an equivalence relation. We formalised a nearly complete, sequential and deterministic sub-language of Erlang with matching logic formulas, used throughout our proofs.

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

4.4 Applicability: a complex case study

In the previous sections, we introduced refactoring oriented programming and the abstractions of the language we designed for specifying Erlang refactorings. We made propositions on whether and how transformations defined in this language are executable or verifiable. Nevertheless, there was another design goal of ours stated at the beginning of the chapter: the language shall be applicable. By this, we mean that useful, complex refactoring transformations should be expressible with the formalism.

Applicability, as well as the methodology of applying decomposition and schemes for defining refactoring is best demonstrated through a meaningful case study. We explain the decomposition process and the role of schemes as building blocks by formally specifying a well-known and fairly complex function refactoring: *generalise function definition.* As object language, we keep using Erlang.

4.4.1 Informal specification

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

1. Generalise the function and then create a fall-back version with the original arity, where the fall-back function simply invokes the newly generalised version by passing as extra argument the extracted expression. This way, call sites to the original function can be left unchanged, since by calling the fall-back function their behaviour remains the same.

2. Change the call sites so that they pass the extracted expression as extra argument to the new, generalised function. This variant does not duplicate the function, but may affect a large number of code locations if there are a number of references to the generalised function.

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

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

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

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

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

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

4.4.2 Decomposition

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

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

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

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

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

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

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

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

```
z(X) -> begin X*fun() -> i() end() end.
f(X) -> z(X).
g(X) -> f(X+1).
```

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

```
z(X) -> begin Y=fun() -> i() end, X*Y() end.
f(X) -> z(X).
g(X) -> f(X+1).
```

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

```
z(X) -> Y=fun() -> i() end, begin X*Y() end.
f(X) -> z(X).
g(X) -> f(X+1).
```

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

```
z(X, Y) -> begin X*Y() end.
f(X) -> z(X, fun() -> i() end).
g(X) -> f(X+1).
```

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

```
f(X, Y) -> begin X*Y() end.
f(X) -> f(X, fun() -> i() end).
g(X) -> f(X+1).
```

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

4.4.3 Formal definition

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

Composite refactoring functions

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

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

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

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

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

Prime refactoring functions

In this case study, all prime refactorings are expressed with schemes. First, we define a local refactoring for making an arbitrary expression "movable", as a result of which the expression is wrapped into a lambda function. Though being a simple change, it has complex conditions: it requires that the expression does not bind any variables that are used outside the expression (predicate `non_bind`), while another condition binds a metavariable to hold the variable names that are free in the expression (referred to by the expression, but bound in the context).

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

We define two instances of the variable introduction scheme for introducing and lifting the new parameter of the generalised function. Both define the syntactic construct creating the binding, determine the place (the scope) of the binding, and they also specify the rewrite rule that will transform the target expression to use the newly introduced binding. Since new variable bindings can be placed either in the current scope or in an outer scope, this has to be decided in the instantiation of the scheme, while the conditions regarding name clash should be handled inherently. With this, we can express both introduction and lifting with the same scheme, and in the second one, the variable name is coming from the already present binding rather than from the refactoring argument.

Syntactic noise (e.g. dead scope) introduced by intermediate steps can be removed by dedicated clean-up refactorings at the end of the process, however, we do not include clean-up transformations in this definition.

```
INTRODUCE VARIABLE
  extract_to_variable(Name)

DEFINITION IN SCOPE
    Name = E

REFERENCE
    E
    ----
    Name
```

```
INTRODUCE VARIABLE
  outer_variable()

DEFINITION IN OUTER SCOPE
    Name = E

REFERENCE
    Name = E
    --------
    Name
```

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

```
INTRODUCE FUNCTION extract_to_function(Name, Params..)

DEFINITION
    Name(Params..) -> E .

REFERENCE
    E
    --------------
    Name(Params..)
WHEN is_subset(free_vars(E), vars(Params..))
```

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

```
FUNCTION REFACTORING var_to_param(X)

DEFINITION
    (Args..) -> X = E, Body..
    -------------------------
    (Args.., X) -> Body..

REFERENCE
    (Args2..)
    ------------
    (Args2.., E)
WHEN pure(E) AND closed(E)
```

A special case of function refactoring is function signature refactoring, which only transforms the head of the function definition and its references. As we demonstrated in our previous paper [29], this scheme can be used as well for renaming a function and to restructure or reorder its arguments.

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

4.4.4 Verification

Since all refactoring functions used in the compound refactoring function are instances of refactoring schemes, the entire definition of “generalise function” presented above can be automatically verified (see Proposition 4.3.4). The verification of some schemes and their instances used in this case study have sketch proofs in our paper [30].

4.5 Summary

In this chapter, I presented a formalism that allows for specifying refactoring transformations in an executable and verifiable way. The proposed method is based on an enhanced variant of conditional term rewriting on semantic program graphs. Complex, extensive context-sensitive transformations are made automatically verifiable via semantics-driven refactoring schemes, which ensure completeness and consistency of the multiple changes in the program.

Thesis 2. *I have developed scheme-based term rewriting with semantic strategies and semantic conditions executed on semantic program graphs, and showed that this novel approach provides an effective system for defining correct-by-construction refactoring. Erlang refactoring definitions specified in this method via decomposition to scheme instances can be automatically verified and interpreted in an Erlang static analysis and transformation tool. I have specified several complex refactorings with this method to demonstrate its applicability.*

I believe that with this work I established the basics of refactoring oriented programming, which may have influence on how semantics-preserving transformations are specified formally. Even though the presentation of the results in dissertation is partly Erlang-specific, the methodology can be adapted to other languages. Indeed, one of my fellows is already working on making the method available for Java refactoring specifications.

Future work

There are different aspects these results shall be improved. On one hand, I will prepare even more case studies to examine the limits and to demonstrate the capabilities of the method. Furthermore, the implementation has to be made stable and open for contribution, this is a definite requirement if we want to make it popular among people in the field.

Another aspect that has to be addressed is generality. In order to spread the idea, I will need to make sure the method is realised for different, widely used programming languages, like Java or C++. As I mentioned already, the prototype design and a draft paper explaining it is already available for the Java language.

Last but not least, sooner or later, together with my fellows we will need to get rid of trusted components of the framework and start building verification methods for the entire refactoring system. Only this achievement can ensure that the refactoring is not only trustworthy, but correct.

5 Extending languages via program transformations

"If someone claims to have the perfect programming language, he is either a fool or a salesman or both." (Bjarne Stroustrup)

No programming language is perfect, but we opt for one or the other based on their unique features that make them the best choice for solving our particular problem. For instance, applications written in Erlang are famous for their robustness, fault-tolerance and scalability, which may be definite and understandable reasons to choose Erlang to implement a wide variety of systems. On the other hand, many programmers are dissatisfied with the syntax of the language, the lack of static checks and the limits of extendability.

In case of such well-established languages, language extensions are incorporated slowly, with a high attention on preserving stability; in exchange, usually they provide a fairly easy way to add ad-hoc extensions to the language via compiler plugins manipulating the syntax tree or the intermediate representation. Erlang, along with its compiler, was not designed with flexible extensibility in mind: although the compiler supports compile-time syntax tree transformations via the so-called 'parse transformations', the use of these is extremely limited.

If one wants to add features, or just complex syntactic sugars to Erlang, the only viable option is creating a new compiler or a pre-compiler for the extended language (not counting the possibility of forking the official one). Creating such a system requires implementing the entire compiler architecture, from syntactic analysis to semantic analysis, modelling, transformation and synthesis. This can be achieved with a standalone implementation or in a dedicated language workbench, but it takes a lot of effort for sure.

Problem statement

In one of our projects in 2013, we needed to implement an iTask-like [59] work-flow system in Erlang, but we could not fit the task operations and the remote execution into the language. Neither the syntax nor the semantics was proper. At the same time, we found that there is a list [17] of plenty of similar possible language extensions edited by the Erlang community. Most of these enhancement proposals are simple extensions to the syntax and more or less complex extensions to the semantics. They cannot be implemented by ordinary parse transformations, they need changes in the compiler, or literally creating a new compiler.

Interestingly enough, just like the language features we needed for the work-flow embedding, the Erlang enhancement proposals could be realised as transformations to the original language by translation. We did not want to put our hand on the official compiler, but we were in possession of our own Erlang analysis and refactoring framework, which implements the components the compiler does. We started to investigate if we can reuse the refactoring engine to implement language extensions via some kind of translation.

What is the pragmatics of such a methodology? Can we translate all the required extensions to the original language? We are going to address these questions in this chapter. We propose a method that allows for adding extensions to the language without modifying a single line of its compiler. Our approach in some sense is similar to hygienic macros, but on semantic program graphs, and implemented by a pre-compiler built upon a refactoring tool [6]. The method allows for extending Erlang with complex, novel features by defining their translation semantics.

Structure of this chapter

The chapter is structured as follows. First we overview related results in Section 5.1. Section 5.2 explains how we employ a refactoring engine to serve as a pre-compiler (translator) for the extended language, and then Sections 5.3 and 5.4 demonstrate two complex language features implemented with the proposed method. Finally, Section 5.5 sums up the results and identifies potential directions of improvement.

5.1 Related work

We summarise related results in regard to language extension implementations in various programming languages, particularly in Erlang.

Extensions to Erlang. Erlang has a macro language, and it offers its programmers a way to perform compile-time transformations on the syntax tree of their programs before the actual compilation process. Numerous language extensions have been made with such tree transformations [21, 44], even some query languages in the standard library, but this method has some inherent disadvantages and limitations. So-called parse transformations cannot extend the language syntactically (i.e. only language extensions compatible with the standard Erlang syntax specification are supported). A parse transformation is allowed to change the semantics of already existing language constructs, and to correct programs that otherwise would not compile due to static semantic errors, but it cannot introduce syntactically new constructs. Also, these transformations are executed before any semantic analysis was done by the compiler; thus, the implementation of a parse transformation needs to gather all the semantic information needed on its own, by traversing the tree. Parse transformations are good for making local changes, but implementation of program-wide (or application level) transformations is not practical.

Erlang-based extensible languages. There are programming languages that can be compiled to Erlang, or to bytecode executable by the Erlang virtual machine. They do not allow us to extend Erlang, but they make it possible to exploit the unique capabilities of the Erlang VM through different constructs in their own language. The most famous such language is called Elixir [74]. It is a meta-programming language on Erlang, which means it compiles to Erlang bytecode and offers everything that Erlang does.

Elixir uses a fairly different syntax compared to Erlang, and implements vehicles for powerful meta-programming. Elixir offers hygienic macros: macro arguments are passed as their representation (let us say, their syntax subtree), while the result of the macro expansion is actually the replacement syntax subtree (and not a text). Amongst the other features of the language, its macro language makes it a good choice for prototyping domain specific languages; nevertheless, implementing semantics-aware macros would be as difficult as in the case of parse transforms, and also, our goal is not to move onto a completely different programming language, as it would make us unable to use the tools we already have for Erlang.

Precompilation. In most cases, language extensions are implemented by introducing a precompiler (or preprocessor) into the compilation process, rather than modifying the compiler of the language. The role of the precompiler is to translate the extended language back to the base language. Such a precompiler can be implemented on different levels of program representations. If the extensions are completely separable and independent of the base language, one can implement a so-called lazy precompilation, where only new language elements are taken into account, the rest of the code is unprocessed (typically, macro preprocessors). Processes of this kind can be effective, but they do not allow the meaning of the new concepts depend on the meaning of the rest of the code written in the base language. Heinlein [25, 26] implemented such a lazy precompiler for C+++, which is a C++ based language with user-defined operators. During the process, new operator constructs are translated back to plain C++ code. Note that this idea is very similar to one of our extension case studies.

Translation. Considerably complex language extensions are implemented by creating complete compilers for the extended language, which translate extended programs back to the base language. Baumgartner and Russo [3] implemented the concept of type signatures in C++ by translating new elements back to C++, but similar approach is presented in [1] to translate C++11 code to C++03 to support legacy compilation environments. Eden [60] extends Haskell with a small set of syntactic constructs for explicit process specification and creation, and translates back to plain Haskell by employing Template Haskell. Broberg [10] extended Haskell with regular expression patterns, the meaning of which — similarly to the previously seen extension — is determined by translating them back to base Haskell.

In addition, there are language extensions that implement extensible variants of well-known programming languages. By building on Stratego/XT, SugarJ [19] offers library-based language extensibility for Java, while SugarHaskell [20] does similar for Haskell. Both variants support definition of extensions, and translate instances of the new language elements back to the base languages. Extending Erlang in such a transformation framework would require the formalisation of the entire language before realising any transformations. This initial investment takes almost as much effort as implementing a source code analyser for Erlang, which in turn we already possess.

Without using an existing transformation framework, one can think of completely custom code representations and transformation definitions. The Java Syntactic Extender [2] presents a standalone solution for extending the language with syntactic sugars, while Nystrom [53] implements an extensible compiler framework for Java. For C++, an extensible analysis and transformation framework is implemented in PUMA [73], based on which they can add aspects to bare C and C++ by translation [65]. Their approach is pretty similar to ours. Similar results are presented by Mihalicza et al. [51], based on another toolset for C++, they integrate an advanced access control in the language by using translation semantics.

Portable functions in Erlang. We remark that following the publication of the papers this chapter is based on, two separate implementations have been created for portable functions in Erlang. One [54] is based on parse transformation, but it is extremely simple, does not even handle function dependencies, the other one [68] is a native implementation in the compiler, with a more advanced closure semantics tracking function dependencies, but this solution has not been integrated into the official compiler either.

5.2 Pre-compilation in a refactoring tool

This section explains the methodology of employing a refactoring system for implementation of language extensions. Apparently, refactoring is not compilation, it uses the same language for the input and the output program. In contrast, a compiler transforms a program of language A to a program in another language B. Yet, if B is a subset of A, the compilation can be understood as a refactoring in language A.

5.2.1 From refactoring to translation

Let us make a clear distinction between program translation and program rephrasing. While during translation the language of the transformed program and the language of the result is different, with program rephrasing the program is turned into another program in the same language. Refactoring is essentially rephrasing the program, while compilation translates the code to a lower level language.

- *Refactoring.* Probably, the most well-known source-to-source program transformation is refactoring [22]. Refactoring does not change the language of the program, it rephrases the code in the very same language, without affecting the meaning, the semantics of the code. Refactoring requires as extensive syntactic and semantic analysis as a compiler does.

$$\begin{tikzcd}\boxed{\text{Erlang}} \arrow[r, "\text{refactoring}"] & \boxed{\text{Erlang}}\end{tikzcd}$$

- *Compilation.* If the abstraction level of the input and the output language is significantly different (usually from higher-level to lower-level), we say that the translation implements compilation. During compilation, we associate elements of the higher-level language with (series of) elements of the lower-level language; this association bridges the abstraction gap between the two languages.

$$\begin{tikzcd}\boxed{\text{Erlang}} \arrow[r, "\text{compilation}"] & \boxed{\text{bytecode}}\end{tikzcd}$$

- *Translation* We define translation as a process that differs from compilation in a sense that even though the languages in question are different, the abstraction gap between them is very small or does not even exist. We will typically use this term for referring to the process of compiling an extended language back to the base language (also called pre-compilation). In our specific setting, the Erlang programming language extended by syntactic sugars and tailored language elements will be translated back to Erlang.

$$\begin{tikzcd}\boxed{\begin{array}{c}\text{Extended}\\\text{Erlang}\end{array}} \arrow[r, "\text{translation}"] & \boxed{\text{Erlang}}\end{tikzcd}$$

Architecture for pre-compilation

A refactoring system is designed to transform programs inside a language. It has to be tailored to be used for translation, but definitely it is feasible. Figure 5.1 indicates that the architecture of a refactoring system and that of a compiler are pretty similar, both perform the following essential tasks of language processors:

- *Input handler*: opens the source file and reads the character sequence
- *Lexical analyser*: converts the series of characters into a series of tokens
- *Preprocessor*: performs transformations on the token stream
- *Syntactic analyser*: turns the series of tokens into a syntax tree
- *Semantic analyser*: reveals properties and relations among syntax tree elements
- *Graph transformation*: performs transformations on the (extended) syntax tree
- *Output handler*: prints the internal representation into textual form

Just like in the compiler, in refactoring, source handling is about loading code into a character sequence. The source code analysis steps are basically the same. There are two main differences: 1) the graph transformation in the compiler is done on the syntax tree, while in the refactoring system the semantic program graph is manipulated, and 2) output handling in the compiler is much more complicated and results in an optimised assembly code, while in refactoring it is basically a simple pretty-printing algorithm.

The reason why refactoring does not function as a compiler is that the language of the input and the output is the same. If we raised the abstraction level of the input language or lowered the abstraction level of the output language, we would get a compiler. Similarly, if the languages were different but of the same abstraction level, it would yield a translator. Pre-compilation is just a case of translation, where the output language is a subset of the input language, which is easily implemented by refactoring. Basically, in order to use a refactoring system as a pre-compiler, all we need is to do is prepare the system to accept and handle the extended language, and to ensure that the programs outputted are always elements of the reduced language.

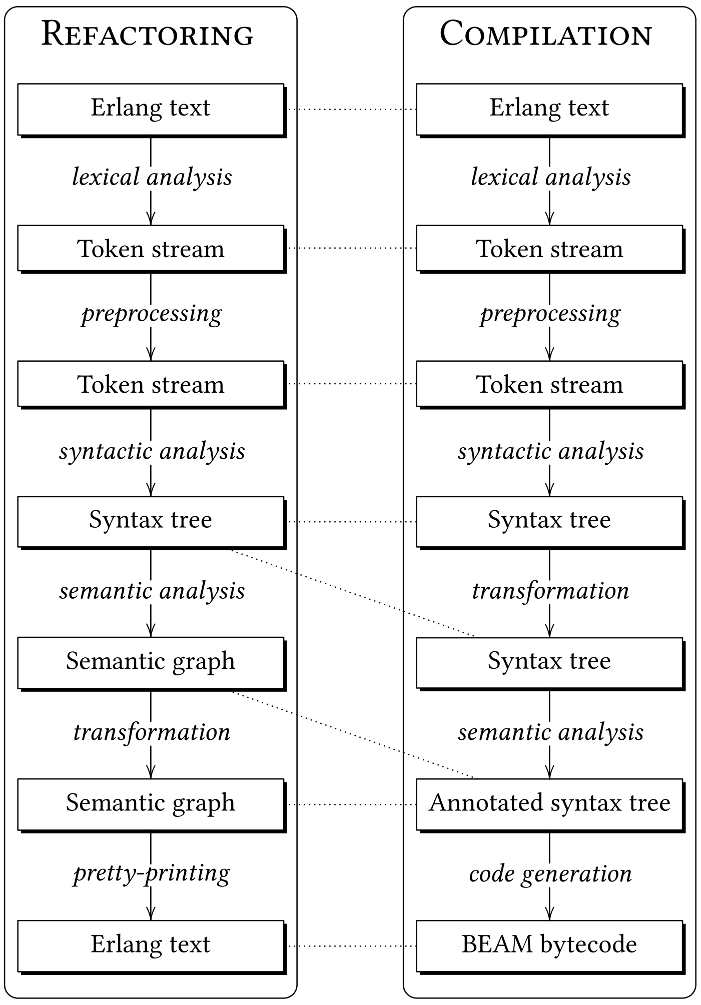


Figure 5.1: Refactoring versus compilation

How do we define the extended language?

Programming languages are defined in terms of three main components: syntax, semantics and pragmatics. From the theoretical point of view, the first two are the most relevant. Syntax determines the set of programs that have a clear meaning in the language (technically, these can be compiled and run). Dynamic semantics determines the meaning, the run-time behaviour of the syntactically-valid, well-formed programs. Usually, programming language semantics is only described informally, the behaviour may depend on the applied compiler and run-time system, but formal definition of semantics is also possible via operational, denotational and axiomatic methods.

When specifying and implementing language features, both syntax and semantics have to be considered. We need to make sure the extended language is accepted by the system, as well as the new language elements have a well-defined semantics. The refactoring system has to be tailored to understand the new language features syntactically, build the corresponding representation, and it has to be able to translate them by means of refactoring-like transformations to the original language of the system.

5.2.2 Extensions and transformations on different levels

Let us enumerate how the refactoring system accepts and translates the extended language for pre-compilation. We will see that the fact that our refactoring system was designed to be easily adjustable and extensible absolutely fits the approach of extending the language it handles. Figure 5.2 shows an overview of the translation process.

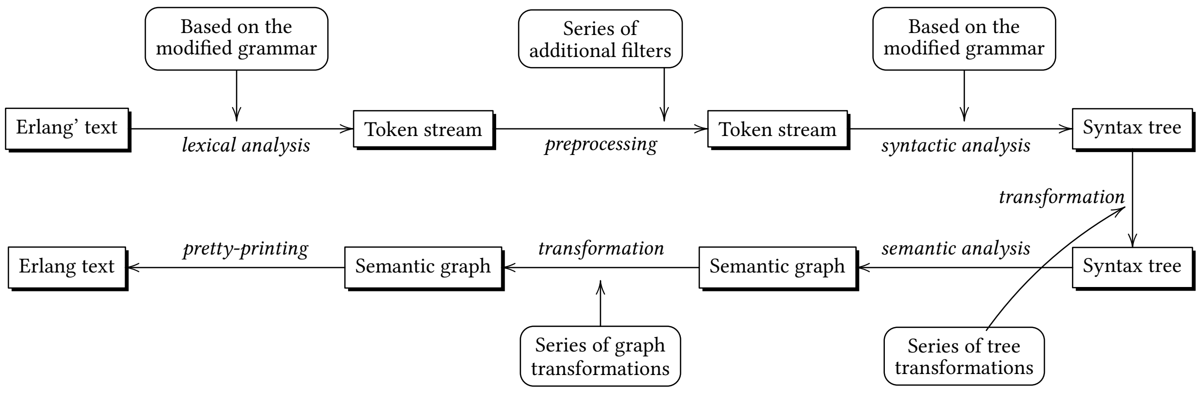


Figure 5.2: The translation process (Erlang' is the extended language)

Lexical layer — extending the scanner

The refactoring system allows for easy definition of additional token classes by specifying a token name and a regular expression for them. Nevertheless, there is a rich set of symbols accepted by base Erlang, thus in most cases there is no need to extend the lexical layer.

Lexical layer — transforming the token stream

The text, after being tokenized by the lexical analyser, goes through the (macro) preprocessor. In ordinary code analysis, it is the preprocessor's responsibility to handle and interpret compiler directives in the code. Specifically in Erlang, the preprocessor tracks macro definitions, expands macro applications and, of course, performs header file inclusion and conditional compilation. In the refactoring system, files (either modules or headers) loaded into the tool are always preprocessed [37], therefore this is the first point where we are allowed to put some customisation into the process.

The implementation of the preprocessor is basically the inspection of the token stream and substitution of those segments that match the format of a compiler directive. Worth mentioning that in the refactoring framework, unlike in the compiler, both the original tokens and the expanded form is stored in order to support fine-grained analysis and precise transformation of code; however, these so-called virtual tokens do not affect our ambitions related to language extensions.

Fortunately enough, it is easy to put additional "filters" (replacement algorithms) on the token stream, modifying the tokens and therefore the input of the syntactic analyser. This can be utilised for turning syntactically invalid code to syntactically valid (according to the grammar of ours), which can be useful if the desired language change would need too complex modifications in the parser.

Example 5.2.1. By using a transformation on the lexical layer, we made bare words available as function and operator names in Erlang.

```
added to(X, Y) -> X + Y.    =>    'added to'(X, Y) -> X + Y.
```

Our additional preprocessing on the token stream looked for subsequent atom constants preceding an opening parenthesis — in Erlang, this can only happen with function definitions and function calls[1], so we can merge those identifiers into one quoted identifier, making the token stream compatible with the syntax.

Example 5.2.2. Another example could be the increment syntax added to the language.

```
f(X) -> X += 10, g(X).    =>    f(X) -> X = X + 10, g(X).
```

In this case, the transformation takes the variable name, the plus sign, as well as the equal sign, and turns it into a proper match expression (used as an assignment). Note that in order to make $+=$ a single operator, it has to be added as a new token.

[1]In later versions of Erlang, type declarations introduced similar syntax, though.

Syntactic layer — extending the parser

Tailoring and introducing language elements is most likely implementable on the syntactic level. For instance, simple syntactic sugars can be added in the parser, and then translated back to the base language as a transformation on the abstract syntax tree.

The refactoring system has its own formal specification of the Erlang language, which merges concrete and abstract syntax, making itself easily extensible. Adding a syntactic construct into the language is as easy as phrasing it in concrete syntax and determining its attributes in the abstract syntax.

```
ECall -> #expr{type=application}
         ( esub->Exp800 esub->EArgList )

EArgList -> #expr{type=arglist}
            ( '(' [esub->Expr {',' esub->Expr}] ')' )
```

Production rules of the context-free grammar, unlike in yacc [33] and its variants, are given in Extended Backus-Naur Form: we do not need to write recursive rules to express the concept of indefinite repetition, rather, repeated symbols are enclosed in curly brackets. Optionality is expressed by putting symbols in square brackets. We followed a similar notation in our generator grammar language in Section 3.3.

The right hand side of a rule starts with a record definition, which identifies the constructor and additional attributes to abstract syntax. RHS symbols are labelled by an atom each, this serves as a label for connecting them to their parent node in the syntax tree (edge labels of terminal symbols are determined by the annotation of their parent node). Worth noticing that the parentheses in apostrophes are terminal symbols, as opposed to those parentheses that encompass right hand side symbols in rules.

Example 5.2.3. This example shows the syntactic rules that have been added to the parser in order to support user-defined operators. We extend the *ECall* and the *EMulOp* categories, to enable usage of both prefix and infix user-defined operators without any parentheses.

```
ECall ->
  #expr{type=application, role=noparen} ( esub->EOperator esub->ExpMax)
  #expr{type=application}               ( esub->Exp800  esub->EArgList)

EOperator ->
  #expr{type=atom, value<-'operator', role=operator} ('operator')

EMulOp ->
  #expr{type=infix_expr, value='/' } (esub->Exp500 '/' esub->Exp600)
| #expr{type=infix_expr, value='*' } (esub->Exp500 '*' esub->Exp600)
...
| #expr{type=infix_expr, value<-'operator', role=operator }
  (esub->Exp500 'operator' esub->Exp600)
```

This modification, along with the previously mentioned token manipulation (Example 5.2.1), will let us write the following expression in our extended Erlang language and get parsed.

```
f() -> 1 added to 2.
```

Note that it is the responsibility of the forthcoming transformations to transform this extended syntax back to bare Erlang.

Syntactic layer — transforming the syntax tree

By extending the parser of the accepted language, we only manage to make the extended language consumable for the refactoring tool, but no transformation is done yet. We need to add some syntax tree transformations in order to translate the new syntax back to bare Erlang, either to implement the translation entirely, or just to ensure the syntax tree be compatible with the Erlang abstract syntax used by the semantic analysers. Steps of this latter kind are very important, since the static semantic analysers are syntax-directed and only work on syntax trees of a fixed schema. If the extended language has a different abstract syntax, it has to be put in order, otherwise semantic analyses will not be able to understand and annotate the tree.

Tree and graph transformations are implemented very similarly, by building on the query and the tree construction libraries (in purely syntactic transformations, semantic queries are not enabled). The nodes to be transformed are selected by graph queries, the replacement is done with the same toolkit. The skeleton of a simple graph transformation is shown in the following snippet: query the node you would like to transform, query all information required to perform the action, then construct the new subtree based on the information gathered, and finally, replace the old subtree with the new one.

```
    Node = query( ... ),
    [Child1, Child2 | _] = query( Node, ... ),
    NewNode = construct( ... Child1 ... Child2 ...),
    replace(Node, NewNode)
```

There can be multiple syntax tree transformations which may contribute to the implementation of several language extensions, and they are executed one after the other, like a pipeline. Note that the order does matter in case if the forms of syntax subtrees affected by different transformations may overlap. Syntax tree transformations cannot rely on semantic information (context-dependent properties), so complex transformations should be postponed to the next phase.

Example 5.2.4. Let us quote a snippet from the transformation implemented for user-defined operators. It transforms the previously introduced syntactic elements (see Example 5.2.3) such that prefix operator applications are turned into proper function applications, while binary operators used in infix notation are re-parsed according to their precedences and associativity rules (these are specified by directives in the extended syntax, in a similar fashion as they are denoted in Haskell).

```
transform(Expr, #expr{type=application, role=noparen}) ->
    [Name, Arg] = ?ESG:path(Expr, [esub]),
    New = ?Syn:construct({app, copy(Name), [copy(Arg)]}),
    replace(Expr, New),
    ?ESG:finalize(),
    children(New);

transform(Expr, #expr{}) ->
    [{Link, Parent}] = ?Syn:parent(Expr),
    case {?ESG:data(Parent), ?ESG:data(Expr)} of
        {#expr{type=infix_expr}, _} ->
            children(Expr);
        {_, #expr{type=infix_expr, value = Op}} when Op /= ':' ->
            I = ?ESG:index(Parent, Link, Expr),
            ?ESG:remove(Parent, Link, Expr),
            Seq = flatten(Expr),
            {ok, NewExpr} = analyse(Seq),
            ?ESG:insert(Parent, {Link, I}, NewExpr),
            ?ESG:finalize(),
            children(NewExpr);
        {_, #expr{}} ->
            children(Expr)
    end;
```

In the above snippet, you can observe how the syntax tree elements are deconstructed and then reconstructed according to the desired change. The manipulation is done on the abstract syntax directly, but the functions *path*, *construct*, *insert* and *replace* simplify tree queries and edits.

The re-parsing logic is done in the functions *flatten* and *analyse*, which first take a sub-expression and flatten it like all operators had the same precedence and associativity, and re-analyse the expression according to the precedence and associativity properties found in the directives afterwards.

Semantic layer — semantic graph

Unlike in syntactic transformations, manipulation of the semantic graph allows us to rely on the semantic properties and relations uncovered by the various semantic analysers (e.g. we can query data-flow and binding information, or potential side-effects). Context-dependent language features can only be implemented at this level, so semantic graph transformations are the point where we are supposed to do actual translation of complex language extensions. Just like before, the manipulation of the graph is realised by graph queries and tree transformations.

Example 5.2.5. As an example to semantic transformations, consider the portable function use case, where the closure has to be coupled with all its dependencies. This needs semantic analysis of bindings, which helps us determine all the functions that a closure may refer to; in the semantic program graph this information is available right after semantic analysis has been done.

```
f(X) -> fun(X) -> g(X) end, ...
g(X) -> ...
```

When transforming the SPG of the above piece of program, simple graph queries can be employed to figure out that the function g is a dependency of the closure defined in f.

5.2.3 Implementing semantics as a transformation

Language extensions should precisely define the syntax and semantics of the features to be added. The implementation of these features, i.e. the translation (or pre-compilation) is realised as a series of conditional graph transformations in the refactoring system, which can also be understood as a series of context-sensitive conditional tree transformations.

Remark. We define these with algorithms manipulating the representation, but observe that these transformations could well be expressed with the transformation language presented in the previous chapter. We did not use the transformation language in this project, because it was designed and developed years later.

Essentially, what we do is adding sugars to the language, meaning the semantics of the new language features have to be expressible in the original language, determining a translation semantics for the new elements. Nevertheless, the features we add are "semantic" or "context-sensitive" sugars. The following sections present case studies that show the capabilities of the method.

Example 5.2.6. Let us consider a simple example of adding increment syntax (same as seen in Example 5.2.2). Looking at it as a transformation pattern, we could conclude with the following translation semantics:

$$S[\![\langle \textit{variable}\rangle\texttt{+=}\langle \textit{expression}\rangle]\!] = [\![\langle \textit{variable}\rangle = \langle \textit{variable}\rangle + \langle \textit{expression}\rangle]\!]$$

Or expressed in our transformation language:

```
   V += E
  ---------
  V = V + E
WHEN var(V)
```

Apparently, the definition of the rewriting (and therefore the translation semantics) is more complex if the context is involved.

Static semantics

Since the transformation is conditional, there may be cases when it is not applicable. In case of refactoring, on such a falsified condition we would conclude that the refactoring cannot be executed in a behaviour-preserving manner. In the setting of language extensions, transformation conditions not met signal static semantic errors.

Take as example the user-defined operators. Operator symbols are always accepted by the lexical and syntactic analysers in the tailored refactoring system, yet if their precedence and associativity is not declared, the translation process needs to report a static semantic error indicating that expressions cannot be translated to the core language. That is, before the real compilation of the translated code, the pre-compiler can detect compilation errors, too.

5.3 Adding user-defined operators to Erlang

The first case study we implemented with our language extension method was adding user-defined operators to Erlang. In this section, we explain in detail the addition of binary operator declarations and their applicability.

The parser grammar of the refactoring system only required the definition of a new token consisting of special symbols, and a new derivation rule that allows infix expressions to be composed with a special operator (see Example 5.2.3). Declarations of the operators are given in terms of ordinary module attributes, where the argument depicts the name and the priority of the new operator symbol.

The translation is implemented as a syntax tree transformation. It looks for infix expressions, flattens them entirely, and then performs operator precedence analysis on the entire corresponding subtree, but according to the precedence and associativity rules defined at the beginning of the transformed file (see Example 5.2.4). The above example of the extended language is translated to the following base Erlang code.

Example 5.3.1. The following code snippet is written in Erlang extended with user-defined operators. We add two operator symbols, *!!* for accessing a list element by its index, and >-< as the list merge operator. Both operators are declared to be left-associative and are of priority 2 and 3, respectively.

```
-module(operator).
-infixl({ !!  , 2 }).
-infixl({ >-< , 3 }).

f(N) -> [1,2,3] >-< [3,4,5] !! N.

!! (L,  I ) -> lists:nth(I, L).
>-<(L1, L2) -> lists:merge(L1, L2).
```

The translation process consumes both the operator declarations and the function definitions. When reaching the definition of function f, it flattens and re-analyses the infix expression so that the evaluation order is disambiguated based on the relative precedence of the operators (merge binds tighter than indexing). The infix syntax is replaced by ordinary function calls, and the operator declarations are dropped from the final result.

```
-module(operator).

f(N) -> '!!'('>-<'([1,2,3], [3,4,5]), N).

'!!' (L,  I ) -> lists:nth(I, L).
'>-<'(L1, L2) -> lists:merge(L1, L2).
```

It is worth mentioning that the translation also takes care of the operator definitions: the operator symbols are enclosed in apostrophes in order to make them legal function names in Erlang.

We note that the method could be used to redefine the semantics of built-in operators, although the current implementation is not prepared for that case. Nevertheless, it is absolutely possible to identify if an Erlang operator is being redefined by the transformation, and the re-parsing process can map (transform) the built-in operator to the function giving new semantics to the symbol.

Bare words operators. As mentioned already in Example 5.2.1, with simple manipulation of the token stream, we can enable the use of operators composed of bare words. With this, we can achieve a cool feature which provides even more readable and well-embedded domain specific code: by combining operator names and variable names, one can compose expressions looking like sentences. The only extra transformation here is adding a step to the standard pre-processing phase.

Example 5.3.2. Another example of custom operators in Erlang is demonstrated with bare words. We introduce two operator symbols, *shows* and *added to* (this latter one consists of two atom literals).

```
-module(atom_operator).
-infixl({ shows    , 3 }).
-infixl({ added to , 4 }).

f(A, B) -> standard_io shows A added to B.

added to (A, B) -> A + B.
shows    (D, A) -> io:format(D, "~p~n", [A]).
```

The transformation is very similar to the previous example, but in addition, the translation makes sure that the operator composed of multiple words is handled as one single operator symbol.

```
-module(atom_operator).

f(A, B) -> shows(standard_io, 'added to'(A, B)).

'added to' (A, B) -> A+ B.
shows      (D, A) -> io:format(D, "~p~n", [A]).
```

Implementation. Essentially, in the transformation process, infix expressions composed with user-defined operators are turned into function calls, and the operator names are put in quotes in order to turn them into legal atom literals.

You can see that this way we can embed simple domain specific languages, which in turn can rely on the well-known features of Erlang. On the other hand, note that since Erlang is dynamically typed, the embedded language will be dynamically typed as well, unless one implements domain specific concepts and their constraints on the transformation level rather than as special operators. For more details on how we implemented a work-flow DSL with special operators in Erlang, we refer to our paper [39].

5.4 Implementing code migration in Erlang

The real strength of using the refactoring framework for implementing translations shows off when dealing with a concept like portable functions (code migration). This requires the transformation to be aware of semantic dependencies among functions, variables and type declarations.

In Erlang, functions are first-class objects, can be stored in variables, can be arguments or return values of other, higher-order functions. Anonymous functions can be defined inside normal functions, and therefore create closures by referring to names bound in their function context. Function closures in Erlang behave similarly to those in most mainstream programming languages: the lexical variable scope along with a reference to the function code gives the denotation of the closure.

Function closures provide a simple solution to passing code or computation around in the program, while the free variables of the closure are taken from the location where the closure has been created. One can even send a function closure over the network by using standard Erlang message passing, without any errors or warnings. However, on the receiving node of the network, the function reference stored in the closure object is very probably invalid and causes a run-time exception. Practically, function closures cannot be sent between nodes of the network in Erlang. On the other hand, code migration is an essential feature of distributed computing.

Example 5.4.1 (Sending a standard closure to another network node)**.** The following example outlines the limitations of standard closures in Erlang. We cannot send functions over the network from one node to another, because function references on one node are not likely to be valid on other nodes.

```
-module(sender). % running on node 1
main() -> receiver ! fun() -> "computations" end. % sending a closure
main() -> remote_receiver ! fun() -> "computations" end.
```

When the function is sent from one process to another process running at the same node, no error or exception happens, because processes at a node share the same function reference table.

```
-module(receiver). % running on node 1
main() -> receive F -> F() end. % "computations"
```

However, if the receiver process is at a remote node (say, node 2), the application of the function reference results in an error, as the reference is invalid at the remote node.

```
-module(remote_receiver). % running on node 2
main() -> receive F -> F() end. % causes run-time error
```

Example 5.4.2 (Dependencies)**.** Function closures may refer to variables, functions, types etc. When sending them over the network as data, we need to make sure that all the dependencies are handled properly and will be available at the receiving side. What are the dependencies we need to track?

```
f(X) ->
  Y = g(X),
  F = fun!(X) -> h(X)*Y end,
  self() ! F,
  receive
    Fun -> io:format("~p", [!Fun(X)])
  end.
```

Even in this simple example the transformation should discover that the function F depends on function h (which may in turn depend on other functions that should be ported as well), and also that the variable Y is freely occurring in the body and thus requires special care. On the other hand, X is a variable of the closure itself, it does not belong to the context to be ported.

5.4.1 The portable closure semantics

In order to make function closures portable, we need to implement a closure semantics that encapsulates the code of the function, not only a node-local reference to it, along with all the dependencies of the function. We can do this in compilation-time, by constructing the function closure by taking its syntax tree and coupling it with the well-defined context. In this setting, context has to be carefully considered, because if the function closure to be sent refers to other functions or types that are not available at the receiving side, those have to be regarded as part of the context and sent along.

Creating a closure

The denotation in our closure semantics is a complex data type that consist of the code of a module (rather than a single function), the name of the module, and the variable context of the closure. The module contains the code of the closure, as well as the code of all its dependencies. The module name is determined by the hash of its code.

Suppose that we have the following closure definition:

```
fun!(Z) -> X = 1, other_func(X, Y, Z) end
```

We can create a closure denotation from this, which will define an entire module that can be loaded to other Erlang run-time systems. The synthesised module can be sent over the network in binary as a byte-code, or as a bare abstract syntax tree.

The above closure would be mapped to the following tuple when using parsing to abstract syntax tree as serialisation. As it is apparent, the main function to be called in the synthesised module is *ported_fun*, and it inherited the original arguments of the closure. The variable X was defined in the closure, so it is not ported, variable Y on the other hand was a free variable, therefore, it is encoded in the closure semantics.

```
{"-module('86431211').
  ported_fun(Y) -> fun(Z) -> X = 1, other_func(X, Y, Z) end.
  other_func(X, Y, Z) -> ...
  ",
 [Y], '86431211'}
```

Using binary encoding, we would pre-compile the synthesised module and send it as bytecode over the network. The rest of the tuple is not affected.

```
{binary:encode_unsigned(
    20317596335189431643381106338993842114797618779509410959 [...]),
 [Y], '86431211'}
```

Note that the unsigned number in the last line represents an arbitrarily large binary string, which is the compiled form of the above source code. The advantage of sending the binary format is obtaining a kind of obfuscation for the ported code; the disadvantage is that the run-time system at the receiving side has to use the same set of operation codes as the sender.

Applying a closure

Closures denoted with the above data type should be able to be executed, therefore we need to tailor the semantics of closure calls as well. We need to extract the module from the denotation, compile (if necessary) and load it into the runtime system, and then call the main function (*ported_fun*) in them. Loading code into the running Erlang system is straightforward thanks to the hot code loading capabilities of the runtime system.

Let us consider the following closure call.

```
PortedFun(2)
```

Supposing that *PortedFun* contains the closure defined above in the binary format, we would apply it with the following snippet:

```
( begin
    {Binary, Context, ModuleName} = PortedFun,
    case code:is_loaded(ModuleName) of
      false -> code:load_binary(ModuleName, ModuleName, Binary);
      _ -> ok
    end,
    erlang:apply(ModuleName, ported_fun, Context)
  end )(2)
```

The closure is divided first into the code, context and module name fragments. The code is loaded into the system, and then the main function is called in the ported module with the original context. This call results a function that mimics the original closure, and can be called with the actual parameters on the receiving side. (In the implementation, this entire task is hidden behind a dedicated de-serialisation function.)

Context dependencies

In order to make sure that the closure shows the very same behaviour everywhere it is called, we execute a thorough binding analysis and collect the dependencies of the closure based upon. This includes variable (value) dependencies, function (code) dependencies and type dependencies.

This is where the refactoring system takes action: this part of the translation is implemented as a semantic graph transformation that can rely on static semantic properties uncovered by the analysis engine. The implementation of the transformation does not have to perform binding analysis manually, it only queries the necessary context-sensitive information.

Variable dependencies. The program model maintained in the refactoring system allows us to make the following queries: which are the variables visible in the body of the closure, which of those are bound in its argument list, and which variables come from the context of the closure.

Example 5.4.3. Analysing the following closure, the refactoring system can tell that variables X, Y and Z are all visible and used in the closure body, yet only Y is free in the closure — this is the variable whose value will be ported as part of the context.

```
fun!(Z) -> X = f(), X+Y+Z end
```

Function dependencies. There is a fine-grained function analysis in the refactoring system, which connects all function references in the syntax to the semantic function object they refer to. This involves connecting function definition sites to function reference sites, which simplifies function dependency lookup.

Example 5.4.4. Analysing the following closure, the refactoring system can tell that the closure depends on functions f and g, and also the *map* function in the *lists* module. This latter, on the other hand, is not identified as a dependency to be ported, because it is part of the standard library in Erlang.

```
fun!(L) -> g(lists:map(fun f/1, L)) end
```

The system is even aware of dynamic function calls, which can only be tracked down by using data-flow analysis combined with function name analysis.

Example 5.4.5. Analysing the following closure, the refactoring system can tell that the closure depends on the function called *foo*, even though its name is determined in the context. Similar, some even more sophisticated dynamic calls and their analysis are addressed in one of our papers [32].

```
foo() -> ...

f() -> F = foo, fun!() -> F() end.
```

Type dependencies. There are type declarations and record definitions in Erlang, which have to be handled as dependencies, too. The translation is prepared to find the type dependencies and copy them to the synthesised module.

Example 5.4.6. Analysing the following closure, the refactoring system can tell that the closure depends on the record type *point*.

```
-record(point, {x,y}).

f() -> fun!() -> #point{x=1,y=1} end.
```

As a result of exploiting the static semantic information extracted and stored in the refactoring framework, we could fairly easily implement a new, portable, dependency-aware closure semantics in Erlang. The translation-based solution is not as efficient as native implementations could be, but this was the first and is still the most advanced answer to the question of (weak) code mobility in Erlang.

5.5 Summary

This chapter demonstrated the applicability of a refactoring framework for easy prototyping of language extensions via employing translation semantics. Since the refactoring system implements very similar functionality and components that of a compiler, it is easily turned into a pre-compiler for an extended language. Building on this idea, we designed translation for two new language elements in Erlang: user-defined operator symbols and portable function closures. The former required an expression re-parsing technique, while the latter was realised by defining a new closure semantics for Erlang anonymous functions. Worth mentioning that I gave the first implementation of code migration in Erlang with my method, which is not the best implementation, but it is lightweight and very effective in terms of dependencies it can handle.

Thesis 3. *I have developed the first implementation of custom operators and code migration in Erlang by employing an Erlang refactoring system as a precompiler for the extended version of the language. Added language features are given a translation semantics to pure Erlang by means of refactoring-like program transformations.*

Future work

The main limitations I can identify regarding this chapter are related to the second case study, implementation of code migration. It is apparent that the "pack everything together" approach is not efficient in terms of space and time, there should be a protocol designed that can be used by the Erlang nodes to negotiate on what dependencies are available on the receiving side. Only those entities (functions, types) should be sent over the network that are not available at the other node (or are of different version), and once transferred, they should not be shared again. This would improve the performance of the solution a lot.

Potential future work of these results is adding more and more language extensions. The Erlang Enhancement Proposal list is growing and growing year by year, and it would be interesting to provide lightweight prototypes for testing whether the proposals make sense and shall be included in the official language. In the future, I will probably seek students to implement extensions in the system.

6 Summary

The dissertation discussed methods for static and dynamic verification of refactoring, as well as semantics-driven pre-compilation via program transformations.

Testing, especially property-based testing, provides a widely adaptable verification technique, supposed that the input domain can be defined by data generators and the correctness property can effectively be checked. I presented a method that allows for synthesising data generators from formal grammars expressed in a concise notation, and I managed to verify Erlang refactoring steps by formalising a subset of Erlang in this notation and synthesising a generator from it. The method is generalisable, any other software with a structured input domain can be subject to L-attribute grammar-based property-based testing.

While testing is a powerful verification technique and can practically be applied to check systems of different complexities, it cannot prove the absence of errors. Only formal verification can prove a refactoring correct, therefore I investigated this verification option, too. I designed a domain specific formalism in which I can specify program transformations in an executable and formally verifiable way. The so-called refactoring language is based on a restricted variant of context-sensitive, strategic term rewriting and the idea of pre-verified transformation schemes. In this language, I specified a number of well-known and useful refactoring steps for Erlang. It is to be carefully investigated what the limits of the expressiveness of this language are.

As a side-effect of the excessive amounts of work with static analysis and program transformation, I also happened to look at special applications of transformations and refactoring. In particular, I tailored the refactoring system to function as a semantics-based pre-compiler for an extended version of Erlang, and I used this method to implement user-defined operators and portable closure semantics. This latter added a very important and powerful feature to the language, enabling code migration among Erlang nodes.

The work presented in this document advances how program transformations are specified, verified and applied for different purposes. I sincerely hope that my results will influence how refactoring transformations are understood and specified in the future.

Bibliography

[1] G. Antal, D. Havas, I. Siket, A. Beszédes, R. Ferenc, and J. Mihalicza. Transforming C++11 Code to C++03 to Support Legacy Compilation Environments. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186, Oct 2016. doi: 10.1109/SCAM.2016.11.

[2] J. Bachrach and K. Playford. The Java syntactic extender (JSE). *SIGPLAN Not.*, 36 (11):31–42, Oct. 2001. ISSN 0362-1340.

[3] G. Baumgartner and V. F. Russo. Implementing signatures for C++. *ACM Trans. Program. Lang. Syst.*, 19(1):153–187, Jan. 1997. ISSN 0164-0925.

[4] I. Bozó, V. Fördős, Z. Horváth, M. Tóth, D. Horpácsi, T. Kozsik, J. Kőszegi, A. Barwell, C. Brown, and K. Hammond. Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, pages 13–23, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3038-1. doi: http://doi.acm.org/10.1145/2633448.2633453.

[5] I. Bozó, V. Fördős, D. Horpácsi, Z. Horváth, T. Kozsik, J. Kőszegi, and M. Tóth. Refactorings to enable parallelization. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming*, pages 104–121, Cham, 2015. Springer International Publishing. ISBN 978-3-319-14675-1.

[6] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, and M. Tóth. RefactorErl – Source code analysis and refactoring in Erlang. In *Proceedings of the Symposium on Programming Languages and Software Tools*, SPLST'11, pages 138–148, Tallinn, Estonia, 2011.

[7] I. Bozó, M. Tóth, M. Tejfel, D. Horpácsi, R. Kitlei, J. Kőszegi, and Z. Horváth. Using impact analysis based knowledge for validating refactoring steps. *Studia Universitatis Babes-Bolyai Informatica Journal*, 56(3):57–64, 2011.

[8] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inf.*, 69(1-2):123–178, July 2005. ISSN 0169-2968.

[9] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.11.003.

[10] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. *SIGPLAN Not.*, 39(9):67–78, Sept. 2004. ISSN 0362-1340.

[11] F. Cesarini and S. Thompson. *Erlang Programming.* O'Reilly Media, Inc., 2009. ISBN 978-0-596-51818-9.

[12] S. Ciobâca. Reducing Partial Equivalence to Partial Correctness. In *Proceedings of SYNASC '14*, pages 164–171. IEEE, 2014. doi: 10.1109/SYNASC.2014.30.

[13] J. Cohen. Renaming global variables in c mechanically proved correct. In G. Hamilton, A. Lisitsa, and A. P. Nemytykh, 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.

[14] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287651. URL http://doi.acm.org/10.1145/1287624.1287651.

[15] D. Drienyovszky, D. Horpácsi, and S. Thompson. Quickchecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 75–80, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0253-1. doi: http://doi.acm.org/10.1145/1863509.1863521. URL http://doi.acm.org/10.1145/1863509.1863521.

[16] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 170–178, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. URL http://dl.acm.org/citation.cfm?id=800078.802529.

[17] eep. Erlang Enhancement Proposals (EEPs). http://www.erlang.org/eeps/, June 2013.

[18] T. Ekman and G. Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42 (10):1–18, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297029. URL http://doi.acm.org/10.1145/1297105.1297029.

[19] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, Oct. 2011. ISSN 0362-1340.

[20] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. *SIGPLAN Not.*, 47(12):149–160, Sept. 2012. ISSN 0362-1340.

[21] G. Fehér and A. G. Békés. ECT: an object-oriented extension to Erlang. In *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, ERLANG '09, pages 51–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-507-9.

[22] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, July 1999. ISBN 0-201-48567-2.

[23] A. Garrido and J. Meseguer. 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.

[24] H.-F. Guo and Z. Qiu. Automatic grammar-based test generation. In H. Yenigün, C. Yilmaz, and A. Ulrich, editors, *Testing Software and Systems*, pages 17–32, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-41707-8.

[25] C. Heinlein. C+++: User-Defined Operator Symbols in C++. In P. Dadam and M. Reichert, editors, *GI Jahrestagung (2)*, volume 51 of *LNI*, pages 459–468. GI, 2004. ISBN 3-88579-380-6.

[26] C. Heinlein. *Concept and Implementation of C+++, an Extension of C++ to Support User-defined Operator Symbols and Control Structures.* Ulmer Informatik-Berichte. Univ., Fak. für Informatik, 2004.

[27] D. M. Hoffman, D. Ly-Gagnon, P. Strooper, and H.-Y. Wang. Grammar-based test generation with yougen. *Softw. Pract. Exper.*, 41(4):427–447, Apr. 2011. ISSN 0038-0644. doi: 10.1002/spe.1017. URL http://dx.doi.org/10.1002/spe.1017.

[28] D. Horpácsi. Extending Erlang by Utilising RefactorErl. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 63–72, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2385-7. doi: 10.1145/2505305.2505314. URL http://doi.acm.org/10.1145/2505305.2505314.

[29] D. Horpácsi, J. Kőszegi, and S. Thompson. Towards Trustworthy Refactoring in Erlang. In G. Hamilton, A. Lisitsa, and A. P. Nemytykh, 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.

[30] D. Horpácsi, J. Kőszegi, and Z. Horváth. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. In A. Lisitsa, A. P. Nemytykh, and M. Proietti, 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.

[31] D. Horpácsi. Attribute grammar for Erlang. http://daniel-h.web.elte.hu/phd.

[32] D. Horpácsi and J. Kőszegi. Static analysis of function calls in erlang: Refining the static function call graph with dynamic call information by using data-flow analysis. *e-Informatica Software Engineering Journal*, 7:65–76, 2013. doi: 10.5277/e-Inf130107.

[33] S. C. Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[34] R. Kitlei. Reconstructing syntax in RefactorErl. PhD workshop, Central Europen Functional Programming Summer School, May 2009.

[35] R. Kitlei, L. Lövei, M. Tóth, Z. Horváth, T. Kozsik, R. Király, I. Bozó, C. Hoch, and D. Horpácsi. Automated syntax manipulation in RefactorErl. In *Proceedings of 14th International Erlang/OTP User Conference*, 2008. URL http://www.erlang.se/euc/08/.

[36] R. Kitlei, L. Lövei, T. Nagy, Z. Horváth, and T. Kozsik. Layout preserving parser for refactoring in Erlang. *Acta Electrotechnica et Informatica*, 9(3):54–63, July 2009.

[37] R. Kitlei, I. Bozó, T. Kozsik, M. Tejfel, and M. Tóth. Analysis of Preprocessor Constructs in Erlang. In *Proceedings of the 9th ACM SIGPLAN Erlang Workshop*, pages 45–55, Baltimore, USA, September 2010.

[38] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005. ISSN 1049-331X. doi: 10.1145/1072997.1073000. URL http://doi.acm.org/10.1145/1072997.1073000.

[39] T. Kozsik, A. Lőrincz, D. Juhász, L. Domoszlai, D. Horpácsi, M. Tóth, and Z. Horváth. Workflow Description in Cyber-Physical Systems. *STUD UNIV BABES-BOLYAI SER INFO*, LVIII(2):20–30, 2013. ISSN 2065-9601.

[40] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4193-6. doi: 10.1145/2934872.2959080. URL http://doi.acm.org/10.1145/2934872.2959080.

[41] A. M. Leitão. A formal pattern language for refactoring of Lisp programs. In *Proceedings of CSMR '02*, pages 186–192, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1438-3. doi: 10.1109/CSMR.2002.995803.

[42] H. Li and S. Thompson. Testing Erlang Refactorings with QuickCheck. In *the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007, LNCS*, pages 182–196, Freiburg, Germany, September 2007. URL http://www.cs.kent.ac.uk/pubs/2007/2648.

[43] H. Li and S. Thompson. A Domain-specific Language for Scripting Refactorings in Erlang. In *Proceedings of FASE'12*, pages 501–515, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28871-5. doi: 10.1007/978-3-642-28872-2_34.

[44] H. Li and S. Thompson. Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 32–39, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1500-5.

[45] L. Lövei, C. Hoch, H. Köllö, T. Nagy, A. Nagyné Víg, D. Horpácsi, R. Kitlei, and R. Király. Refactoring module structure. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 83–89, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: http://doi.acm.org/10.1145/1411273.1411285. URL http://doi.acm.org/10.1145/1411273.1411285.

[46] R. Lämmel, S. Thompson, and M. Kaiser. Programming errors in traversal programs over structured data. *Science of Computer Programming*, 78(10):1770 – 1808, 2013. ISSN 0167-6423. doi: 10.1016/j.scico.2011.11.006.

[47] L. Lövei, L. Hajós, and M. Tóth. Erlang Semantic Query Language. In *Proceeding of 8th International Conference on Applied Informatics, ICAI 2010*, pages 165–172, Eger, Hungary, January 2010. ISBN 978-963-98-94-72-3.

[48] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990. ISSN 0740-7459. doi: 10.1109/52.56422.

[49] P. M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice and Experience*, 22:223–244, 1992.

[50] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution*, 17(4): 247–276, 2005. ISSN 1532-0618. doi: 10.1002/smr.316.

[51] J. Mihalicza, N. Pataki, Z. Porkoláb, and Ádám Sipos. Towards more sophisticated access control. In J. Peltonen, editor, *Proceedings of 11th Symposium on Programming Languages and 7th Nordic Workshop on Model Driven Software Engineering*, pages 117–131, 2009.

[52] U. Norell and A. Gerdes. Attribute Grammars in Erlang. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, Erlang 2015, pages 1–12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3805-9. doi: 10.1145/2804295.2804296. URL http://doi.acm.org/10.1145/2804295.2804296.

[53] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3.

[54] T. Ohta. Portable funs with parse transformations. https://github.com/sile/pfun, 2015.

[55] G. Oláh, D. Horpácsi, T. Kozsik, and M. Tóth. Type inference in Core Erlang to support test data generation. *STUD UNIV BABES-BOLYAI SER INFO*, LIX(1):201–215, 2014. ISSN 2065-9601.

[56] K. Olmos and E. Visser. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules. In R. Bodik, editor, *Compiler Construction*, volume 3443 of *LNCS*, pages 204–220. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25411-9. doi: 10.1007/978-3-540-31985-6_14.

[57] W. F. Opdyke. *Refactoring Object-oriented Frameworks.* PhD thesis, University of Illinois, 1992.

[58] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems*, PLOS '06, page 10, New York, NY, USA, 2006. ACM. ISBN 1-59593-577-0. doi: 10.1145/1215995.1216005.

[59] R. Plasmeijer, P. Achten, and P. Koopman. itasks: Executable specifications of interactive work flow systems for the web. *SIGPLAN Not.*, 42(9):141–152, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1291220.1291174. URL http://doi.acm.org/10.1145/1291220.1291174.

[60] S. Priebe. Preprocessing Eden with Template Haskell. In *Proceedings of the 4th international conference on Generative Programming and Component Engineering*, GPCE'05, pages 357–372, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540291385, 9783540291381.

[61] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, Sep 1972. ISSN 1572-9125. doi: 10.1007/BF01932308. URL https://doi.org/10.1007/BF01932308.

[62] D. B. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois, 1999.

[63] M. Schaefer and O. de Moor. Specifying and Implementing Refactorings. *SIGPLAN Not.*, 45(10):286–301, Oct. 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869485.

[64] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. ISBN 0201656973.

[65] O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. In *Proceedings of the 2005 Conference on New Trends in Software Methodologies, Tools and Techniques*, pages 33–53, Amsterdam, The Netherlands, The Netherlands, 2005. IOS Press. ISBN 1-58603-556-8. URL http://dl.acm.org/citation.cfm?id=1563296.1563301.

[66] N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *Proceedings of PEPM '08*, pages 51–60, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7. doi: 10.1145/1328408.1328417.

[67] M. Tejfel, M. Tóth, I. Bozó, D. Horpácsi, and Z. Horváth. Improving quality of software analyser and transformer tools using specification based testing. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae*, 37:355–368, 2012.

[68] A. F. Tyron Zerafa. Erlang Code Migration. https://github.com/tyronzerafa/Erlang-Code-Migration, 2013.

[69] M. Tóth and I. Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

[70] M. Tóth, I. Bozó, J. Kőszegi, and Z. Horváth. Static Analysis Based Support for Program Comprehension in Erlang. In Acta Electrotechnica et Informatica, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), ISSN 1338-3957 (online), pages 3-10.

[71] M. Tóth, I. Bozó, Z. Horváth, and M. Tejfel. 1st order flow analysis for Erlang. In *Proceedings of 8th Joint Conference on Mathematics and Computer Science*, pages 403–416, Komárno, Slovakia, July 2010. ISBN 978-963-9056-38-1.

[72] M. Tóth, I. Bozó, and Z. Horváth. Reverse engineering of complex software systems via static analysis. Lecture at 4th Central European Functional Programming School, Budapest, Hungary, June 2011.

[73] M. Urban, D. Lohmann, and O. Spinczyk. PUMA: An Aspect-oriented Code Analysis and Manipulation Framework for C and C++. In S. Katz and M. Mezini, editors, *Transactions on Aspect-oriented Software Development VIII*, pages 141–162. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-22030-2. URL http://dl.acm.org/citation.cfm?id=2028556.2028563.

[74] J. Valim. Elixir - A modern approach to programming for the Erlang VM. http://elixir-lang.org/, June 2013.

[75] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A Scripting Language for Refactoring. In *Proceedings of ICSE '06*, pages 172–181, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134311.

[76] E. Visser and Z. Benaissa. A core language for rewriting. *Electr. Notes Theor. Comput. Sci.*, 15:422–441, 1998. doi: 10.1016/S1571-0661(05)80027-1. URL https://doi.org/10.1016/S1571-0661(05)80027-1.

[77] E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Electr. Notes Theor. Comput. Sci.*, 203:103–116, 2008.

Abstract

Program transformations and refactorings are essential elements in software development. From the early days of refactoring, tool support has emerged to provide effortless and trustworthy assistance for quality-improvement, behaviour-preserving transformations of programs. Both academia and industry have great interest in researching and developing static analysis based bug detection and trustworthy behaviour-preserving transformations. Although refactoring tools are getting better day by day, there is room for improvement both in terms of accuracy and reliability.

This dissertation investigates definition and verification methods for program transformations, which can help us build more extensible, more trustworthy and widely applied transformations systems. To advance dynamic verification techniques, a novel notation for L-attribute grammars is introduced, and is given a meaning in terms of data generator functions, which can be used in various ways in property-based testing. The document presents a case study, a grammar for Erlang, and its use in testing an Erlang analysis and transformation system. To address static, formal verification of transformations, a refactoring programming language is proposed as the specification formalism for executable and semi-automatically verifiable refactoring definitions. The language is based on context-sensitive conditional term rewriting, strategy programming and refactoring schemes. Last but not least, as a special application of graph rewriting and tool-assisted program transformation, the dissertation discusses implementation of language pre-compilers in refactoring systems. With this method, a portable closure semantics was added to the Erlang programming language, enabling code migration.

Kivonat

A programtranszformáció és a refaktorálás alapvető elemei a szoftverfejlesztési folyamatnak. A refaktorálást a kezdetektől próbálják szoftvereszközökkel támogatni, amelyek megbízhatóan és hatékonyan valósítják meg a szoftverminőséget javító, a működést nem érintő programtranszformációkat. A statikus elemzésre alapuló hibakeresés és a refaktorálási transzformációk az akadémiában és a kutatás-fejlesztésben is nagy érdeklődésre tartanak számot, ám még ennél is fontosabb a szerepük a nagy bonyolultságú szoftvereket készítő vállalatoknál. Egyre pontosabbak és megbízhatóbbak a szoftverfejlesztést támogató eszközök, de bőven van még min javítani.

A disszertáció olyan definíciós és verifikációs módszereket tárgyal, amelyekkel megbízhatóbb és szélesebb körben használt programtranszformációs eszközöket tudunk készíteni. A dolgozat a statikus és a dinamikus verifikációt is érinti. Elsőként egy újszerű, tömör leíró nyelvet mutat be L-attribútum grammatikákhoz, amelyet tulajdonságalapú teszteléshez használt véletlenszerű adatgenerátorra képezünk le. Ehhez egy esettanulmány társul, amely az Erlang programozási nyelv grammatikáját, majd a teszteléshez való felhasználását mutatja be. A tesztelés mellett a formális helyességbizonyítás kérdését is vizsgáljuk, ehhez bevezetünk egy refaktorálások leírására szolgáló nyelvet, amelyben végrehajtható és automatikusan bizonyítható specifikációkat tudunk megadni. A nyelv környezetfüggő és feltételes termátíráson, stratégiákon és úgynevezett refaktorálási sémákon alapszik. Végül, de nem utolsósorban a programtranszformációk egy speciális alkalmazása kerül bemutatásra, amikor egy refaktoráló keretrendszert előfordítóként használunk a feldolgozott programozási nyelv kiterjesztésére. Utóbbi módszerrel könnyen implementálható az Erlang nyelvben a kódmigráció.

[1]ADATLAP

a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai

A szerző neve: Horpácsi Dániel

MTMT-azonosító: 10029219

A doktori értekezés címe és alcíme: Verification and Application of Program Transformations

DOI-azonosító[2]: 10.15476/ELTE.2018.170

A doktori iskola neve: Informatika Doktori Iskola

A doktori iskolán belüli doktori program neve: Az informatika alapjai és módszertana

A témavezető neve és tudományos fokozata: Horváth Zoltán, PhD

A témavezető munkahelye: Eötvös Loránd Tudományegyetem, Informatikai Kar

II. Nyilatkozatok

1. A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Kulcsár Adinát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: Budapest, 2018. augusztus 30.

[illegible]

a doktori értekezés szerzőjének aláírása

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.

[2] A kari hivatal ügyintézője tölti ki.

[3] A megfelelő szöveg aláhúzandó.

[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.

[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.

[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.