Technical Report: Tosca

The preprOcesSing Compiler generAtor language

Georg Moser
Universität Innsbruck, Austria
georg.moser@uibk.ac.at

Maria A Schett
Universität Innsbruck, Austria
mail@maria-a-schett.net

Lionel Villard
IBM Thomas J. Watson Research Center, USA
villard@us.ibm.com

Disclaimer: Not complete, subject to change—State May, 2016

Abstract. We present Tosca, a compiler generator *and* a programming language. Tosca leaves only a small footprint and we have designed it striving for a light-weight architecture that reduces compiler construction to high-level code generation—rather than relying on a full compilation stack. Given as input the syntax of a source language $\mathcal{L}$ together with its semantics, the Tosca compiler generator generates the source code for a compiler of $\mathcal{L}$—currently in Java 8. After employing an off-the-shelf Java compiler, one obtains an executable compiler for $\mathcal{L}$. The Tosca language, the implementation language of Tosca, is a rule-based, higher-order, lazy functional language that has its roots in higher-order rewriting, more precisely Combinatory Reduction Systems. We successfully used the Tosca compiler generator to generate a compiler for the Tosca language.

1 Introduction

The use of transpilers, or source-to-source compilation, has recently became quite popular especially in the world of JavaScript—with CoffeeScript, Babel, and TypeScript leading the way. The reason is largely due to the fact that the JavaScript language has been slow to evolve and most importantly when new features become available adoption is slow as older browsers do not support them. Transpilers solve both problems. Based on this, and partly motivated by the success of transpiler, we present in this paper Tosca, a generic compiler generator.

Tosca aims to combine two different trends in compiler technology, which have proven to be successful in the past. On one hand, Tosca follows the lead of transpiling and focuses on source-to-source transformation. Here we are attracted by the simplicity and elegance we see in the design of the above mentioned preprocessing languages for JavaScript. In particular, we aspire a light-weight approach to compiler generation, which focuses on high-level code generation. On the other hand, we recognize the fundamental role played by (higher-order) rewriting for various tasks in a compiler. We rewrite source code to its abstract syntax tree in parsing, we normalize (i.e., rewrite) the AST to an intermediate representation, we optimize (i.e., rewrite) the intermediate language in various forms, and finally we rewrite everything in code generation. It was the fundamental idea behind CRSX[1] that this informal use of rewriting can be made precise and even formalized in the formalism of *Combinatory Reduction Systems* (*CRSs* for short). We summarize the contributions of the paper.

1. We present a compiler generator that lifts the concept of transpilers to its full generality. Given a source language $\mathcal{L}$, whose syntax is represented as an ANTLR v4 grammar, and whose semantics is provided in a rule-based form, the Tosca executable emits source code in the target language. After employing an off-the-shelf compiler for the target language, we obtain an executable compiler for $\mathcal{L}$. Our current setup uses JAVA 8 as target language, but C++ or HASKELL could easily well be integrated.

2. For the implementation of Tosca we have designed a rule-based, higher-order, lazy functional language, also called Tosca. The formal core of Tosca are CRSs. The foundation in higher-order rewriting allows for a good coupling of the tasks of source-to-source compilation (parsing, normalizing, generation of code) and their implementation. In particular the semantics of the source language $\mathcal{L}$ is provided in Tosca itself.

3. Finally, Tosca bootstraps. The compiler for Tosca is generated via the Tosca executable.

The compiler generator Tosca and the programming language that comes along are *light-weight.* As a compiler generator it acts like a preprocessor and thus does not rely on a full compilation stack. This entails that code optimization is most of the time passed on to the target language compiler. Similarly, the Tosca leaves a small footprint, as it relies on state-of-the-art features in the design of the target language.

Tosca adds features to general purpose functional programming, some unique and some not, specific to compilers. More precisely Tosca features support for *embedded programs*, *enumeration*, *syntactic variables*, and *specificity-ordered pattern matching.* In particular, the last point is unique. In programming patterns are typically non-ambiguous as the patterns are not overlapping, or made non-ambiguous by fixing an order on the defining equations, e.g., lexical order. Similarly, pattern matching is paramount in compiler, where they are used to accomplish dispatch and code optimization. The latter may even introduce overlapping definitions, as special cases are set aside against a fallback definition. In complex languages it becomes increasingly hard to identify missing or redundant cases, without hampering code optimization.

[1]cf. crsx.org.

Specificity-ordered pattern matching overcomes this issue to some degree. Instead of relying on an ad-hoc lexical order of the rule definitions to disambiguate, we incorporate Kennaway's *specificity* rule [12] and lift it to higher-order programming, employing higher-order unification for patterns. The specificity idea demands that any function definition can only be employed if there is no alternative rule which is more specific. The latter can be verified by the generation of a *specificity tree* that allows to disambiguate between overlapping definitions. In particular this process allows to recognize the fallback option easily, as long as we do not have to disambiguate within equal levels of specificity. In the latter case, the specificity rule is not applicable and an exception is thrown.

The remainder is structured as follows. In the next section we provide more background information and give an informal presentation of the bells and whistles of Tosca. We also introduce Mini ML as a source language that will be used as a running example to demonstrate aforementioned bells and whistles. In Section 3, we provides an informal introduction to CRSs and the translation of this foundation to Tosca Core and eventually to the full Tosca language. Section 4 describes the architecture underlying Tosca. In Section 5, we explain how we can use and used Tosca—for real. In particular we describe the bootstrapping process. Finally we conclude and present future work in Section 6.

The paper occasionally refers to the actual code of the implementation for reference. The reader may find the source (in anonymized form) in the supplementary material for this submission.

2 Background and Informal Presentation

Combinatory Reduction Systems (CRSs), introduced by Aczel [1] and later refined by Klop [13], form a generalization of first-order term rewriting to higher-order rewriting. CRSs provide the theoretical basis of our compiler generator Tosca. It has turned out that higher-order rewriting provides a good match for the various tasks (parsing, normalization, specificity, and code generation) required in a compiler, and thus in its construction.

On one hand Tosca is a compiler generator, that is, *a tool* for constructing compilers, whose setup is depicted in Figure 1 below and detailed in Section 2.1. On the other hand Tosca is a *programming language*, namely a self-hosted, lazy, higher-order functional language, with sometimes unique features, specific to compilers, like specificity-ordered pattern matching, and embedded programs, cf. Section 2.2.

CRSs form the essential core of Tosca. While practical necessities for compiler construction naturally push certain demands on the design (cf. Section 2.3), we have been careful to protect this theoretical underpinning as a separate part (Tosca Core) of our language.

The overall setup of Tosca strives for a light-weight architecture. For example a typical task of a compiler is to optimize the previously generated intermediate representation of the source language. Hence, it may be to be expected that a similar optimization stage is required in a compiler generator to trigger this behavior. However, an elegant possibility is to learn from transpilers. Tosca merely *transforms* the provided source language $\mathcal{L}$ into a dedicated target (intermediate) source representation. Then an off-the-shelf compiler for this target language (for the moment Java) handles the optimization.

The development of Tosca has been greatly motivated by a preceding compiler generator project based on higher-order rewriting, namely CRSX, mainly developed by Rose [19, 20]. CRSX has been successfully applied in practice by IBM DataPower Gateway[2] to implement an XQuery [23] and a JSONiq [7] compiler and is also used as a back-end for the teaching interface HACS[3].[21]

2.1 Generating a Compiler With Tosca

Tosca's main purpose is to ease the construction of compilers. Thus, we start by describing how to construct a compiler with Tosca. Given some programming language $\mathcal{L}$, and some program written in $\mathcal{L}$ which we want to compile: How can we use Tosca to create such a compiler? We depict the overall setup in Figure 1.

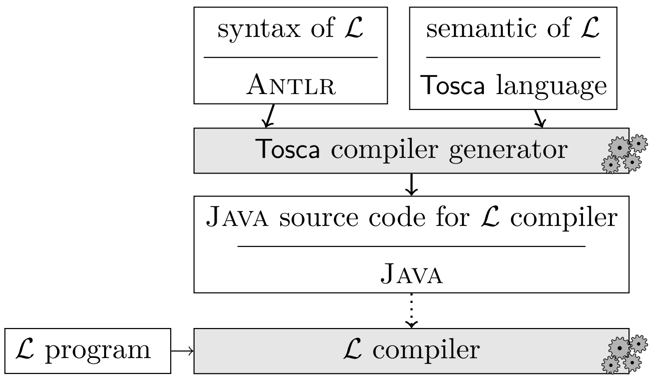

Figure 1: Compiler Generation for Programming Language $\mathcal{L}$.

As can be seen on the top, to create a compiler for $\mathcal{L}$, Tosca needs as input:

1. the *syntax* of $\mathcal{L}$ in the ANTLR v4 format [17], and
2. the *semantics* of $\mathcal{L}$ as Tosca rules.

Based on this input Tosca creates the source code of a compiler for $\mathcal{L}$. In Figure 1 this execution of Tosca is indicated by cogwheels. Currently, the generated source code is JAVA 8 source code. However, the architecture of Tosca allows a simple integration of any suitable target language, cf. Section 4. Finally, the generated JAVA source code has to be compiled—using an off-the-shelf JAVA compiler—to obtain an executable $\mathcal{L}$ compiler. Observe that the $\mathcal{L}$ compiler itself targets a programming language and it is up to the compiler writers to decide which language to use (in the semantics of $\mathcal{L}$).

We exemplify this setup with respect to a simple, strict, and pure functional language, namely MINI ML[4]. We describe the creation of an interpreter for MINI ML, that is, $\mathcal{L}$

[2]cf. www.ibm.com/datapower.

[3]cf. github.com/crsx/hacs.

[4]The syntax is adapted from `andrej.com/plzoo/html/miniml.html`. The full ANTLR grammar is available at `MiniML.g4`.

== MINI ML. First we need an ANTLR grammar describing the syntax, which we will present next. For presenting ANTLR grammars throughout this paper, we will employ the following conventions: Non-terminals, i.e., ANTLR grammar rules, are written in lower case letters, e.g., `expr`. For terminal symbols it depends. If they have a verbatim match, such as keywords and delimiters, we inline and enclose them in `''`, e.g., `'let'` or `'='`. If they cannot be described verbatim, as for example numbers or variables, then we write them in capital letters, e.g., `VAR`.

Example 1. An ANTLR description of the MINI ML language:

```
toplevel : 'let' VAR '=' expr ';;' toplevel ';;'
           | expr ';;'

expr : timesExpr '+' expr | timesExpr '-' expr
      | timesExpr

timesExpr : comprExpr '*' timesExpr | comprExpr

comprExpr : primaryExpr '<' comprExpr
          | primaryExpr '=' comprExpr
          | primaryExpr

primaryExpr : app_expr | '-' INT |
            | 'if' expr 'then' expr 'else' expr
            | 'fun' VAR '(:' ty '):' ty 'is' expr

app_expr : simple_expr | simple_expr app_expr

simple_expr : 'true' | 'false' | '(' expr ')'
            | INT | VAR

tyPrimary : 'bool' | 'int' | '(' ty ')'

ty : tyPrimary | tyPrimary '->' ty

VAR : ['a'-'z' 'A'-'Z']+
INT : ['0'-'9']+
```

Given the syntax of MINI ML, we now need to describe the *semantics* in the Tosca language to generate a MINI ML interpreter. To give an idea and a glimpse of the Tosca language, we look at how we can describe the `'+'`.

```
rule EvalExpr(expr⟦ #timesExpr + #expr ⟧)
  → Plus(EvalTimesExpr(#timesExpr), EvalExpr(#expr))
```

To evaluate the expression `#timesExpr + #expr`, we rely on `Plus` provided by a standard library in Tosca. Here also we already present the feature of *embedded program* snippets: within `expr⟦ ⟧` we can embed MINI ML syntax—a feature explained in detail in the next section. Of course, we could as well define the functionality easily ourselves. Next we look at a different snippet: the `'if␣then␣else'`.

Example 2. The conditional `'if␣then␣else'` is implemented as follows:

```
rule EvalExpr(expr⟦if #expr1 then #expr2 else #expr3⟧)
  → EvalIf(EvalBoolExpr(#expr1), #expr2, #expr3)

rule EvalIf(TRUE, #expr2, #expr3) → EvalExpr(#expr2)
rule EvalIf(FALSE, #expr2, #expr3) → EvalExpr(#expr3)

rule EvalBoolExpr(simple_expr⟦ true ⟧) → TRUE
rule EvalBoolExpr(simple_expr⟦ false ⟧) → FALSE
...
```

The above rules are, of course, incomplete, as indicated by the ellipses. More of Mini ML code snippets will return throughout the next section, where we introduce the Tosca language. Also notice that the choice of showing the Mini ML interpreter as opposed as the compiler is purely driven by presentation considerations, and not technical ones.

Naturally, Mini ML is a toy language. But we have used Tosca already in a real-world application: In Section 5 we describe how we successfully used Tosca to create a compiler for the Tosca language itself, that is, we can bootstrap (aka self-host) Tosca. I.e., the setup depicted in Figure 1 is applicable in its fullest generality, namely we have:

$$\mathcal{L} == \text{Tosca language.}$$

2.2 Tosca as a Functional Language

This section gives an overview of the Tosca language through code snippets. A subset of the formal syntax is given in Section 3.3 and the complete Tosca syntax written as an Antlr v4 grammar can be found in `Tosca.g4`. Tosca is a higher-order, lazy functional language with good support for efficient recursion which makes it easy to work with trees. In particular it is very suited to work on one of the main data structures in compiler construction: *Abstract Syntax Trees* (*ASTs* for short). It naturally borrows many characteristics and features found in existing functional programming languages, such as referential transparency, first-class functions, and pattern matching. Describing them in detail is out of scope. Instead the syntax of the most commonly used features is illustrated in the example below.

Example 3. We give the Tosca syntax for the implementation of `Map` over a list of elements of type `a`.

```
func Map<a b>([a] -> b, List<a>) -> List<b>

rule Map([x] -> #F(x), ())
  → ()

rule Map([x] -> #F(x), (#Yhd, #Ytl...))
  → (#F(#Yhd), Map([x] -> #F(x), #Ytl))
```

A *function* is declared using the keyword `func` followed by its name, parameter types, and return type. In this example, `Map` is a parameterized function with two type parameters `a` and `b`, and it takes two parameters, the first one is a function taking one argument of type `a` and returning a value of type `b`, the second one is a list of values of type `b`. In Tosca, function names and types start with an upper case character. Type parameters and variables start with a lower case character.

A *rule* starts with the keyword `rule`. As usual a rule provides a function definition for the function on the left hand by providing a directed defining equation. Two rules are associated to the `Map` function declaration. For both rules, the first argument `[x] -> #F(x)` matches any function taking one parameter.

We emphasize that the expression `[x] -> #F(x)` is used in Tosca code differently based on context. In the function declaration it plays its standard role as an arrow type. However, perhaps unexpectedly in the first rule it reappears. In this latter context `[x] -> #F(x)` denotes the lambda abstraction `[x]#F(x)` presented in Section 3.1. We allow for this overloading, as the arrow notation is a common notation for closures in C-like languages. However, note that it is merely syntactic sugar which is represented as expected in Tosca Core (cf. Section 3.3).

Continuing with Example 3, matching values are bound to *meta variables*. Meta variables start in Tosca with the character `#`, as `#F, #Yhd, #Ytl` in Example 3. Meta variables which are bound in a pattern can either be discarded, like in the first rule, or used in the rule expression. The matching function in both rules is bound to the meta variable named `#F` (cf. Section 3.1). The second argument of the first rule matches the empty list, i.e., `()`. The second argument of the second rule matches a list with at least one element, bound to `#Yhd` and the rest of the list, using the ... notation, is bound to `#Ytl`.

By default, rules are *applied* in lexical order, from top to bottom, unless another evaluation strategy is specified. In the example, the first rule matches when the second argument is the empty list, and produces the empty list as the result. When the second rule matches, for non-empty list, it produces a list where the first item is the result of the function `#F` applied to the argument `#Yhd`. The list tail is the result of `Map` applied to the rest of the list `#Ytl`. Section 4.4 explains how the default evaluation order is implemented. Sections 2.3 and 4.3 describe howto enhance this functionality to make it more practical.

Finally, we apply `Map` on a real input. Therefore we provide a third rule, a `Main` function.

Example 4 (continued from Example 3)**.** We get:

```
func Main -> List<Int>
rule Main
  → Map( [x] -> Plus(1, x), (1, 2, 3) )
```

The function `Main` shows how `Map` is called with a the function `Plus` and the list `(1,2,3)`. As expected the result of `Main`, when evaluated, is `(2,3,4)`

2.3 Tosca Language Features For Compilers

In addition to these general features, Tosca adds features specific to compilers. Some are rarely found in other programming languages, namely embedded programs, syntactic variables, and specificity-ordered pattern matching, while the remaining one, enumeration, is ubiquitous.

Embedded programs. One of the cornerstone Tosca features is the ability to embed program fragments written in the source programming language $\mathcal{L}$ directly into Tosca programs: inside rules.

As described in Section 2.1, Tosca takes two inputs: the syntax and the semantics of a language $\mathcal{L}$. The syntax is needed for two reasons: first, as mentioned before, to parse input program written in $\mathcal{L}$, but secondly also to parse program fragments embedded in the semantic description of $\mathcal{L}$.

Example 5. Consider this rule:

```
rule InlineLetSimpleExpr(
  toplevel⟦ let #VAR = #simple_expr ;; #toplevel ;; ⟧)
  → ...
```

The special construct `toplevel⟦... ⟧` tells the Tosca compiler to parse the embedded program using the grammar rule named `toplevel`. Section 4.2 gives an in-depth description on how this works.

Example 6. The rule after expanding the embedded syntax

```
rule InlineLetSimpleExpr( Let( #VAR,
    TimeExpr(CompExpr(PrimaryExpr(#simple_expr))),
                             #toplevel) )
  → ...
```

There are several advantages to this approach. First, it makes Tosca programs more readable—as demonstrated in the examples above—by reducing the code size significantly. Also, the correspondence between language syntax and its semantics becomes more apparent. Furthermore, embedded program fragments are sometimes more resilient to some language syntax refactorings. For instance, augmenting the MINI ML language with a new kind of expressions, such as logical operators, has no effect on the rule shown in Example 5, whereas it changes the rule shown in Example 6. On the other hand, changing `';;'` to `';'` has the opposite effect, since it impacts the concrete syntax of the language. Nonetheless, we argue that improved readability prevails over minor language updates.

Enumeration. *Enumeration* with associated values is the principal Tosca data type. In the literature enumerations are also called sum type, tagged union, variant type or recently just enumerations, like in Swift or Rust.[5] It is well-established that enumeration

[5]cf. developer.apple.com/swift and www.rust-lang.org, respectively.

is very well-suited for representing programming language syntax. Following the current trend, Tosca enumerations are declared using the keyword `enum`. Value variants are separated by the character `|`. To show the correspondence between language syntax and enumeration recall the ANTLR grammar for the non-terminal `simple_expr` defined in Example 1:

```
simple_expr : VAR | 'true' | 'false' | INT
            | '(' expr ')'
```

I.e., a simple expression consists of either a variable represented by the lexical token `VAR`, the literals `'true'` or `'false'`, an integer or an `expr` surrounded by parenthesis.

Example 7. This grammar naturally maps to the following enumeration:

```
enum Simple_expr | VAR | TRUE | FALSE
                 | INT | Expr
```

Notice the enumeration value names are slightly changed in order to accommodate the Tosca syntax. Grammar literals are mapped to all upper case name, dropping the surrounding quotes. Section 4.1 formalizes this mapping from ANTLR grammar onto Tosca.

Syntactic Variables. Programming languages commonly define ways to declare variables with strict scoping rules associated to them. Compiler writers rely on operations to determine when variables are not used, e.g., for dead code elimination, or used only once, e.g., for in-lining, or for other purposes. Tosca provides functionalities to deal with *syntactic variables* and *scoped syntactic variables.* Tosca represents either kind of variables the same way as variable (cf. Section 2.2): by an identifier starting with a lower case character. For instance `var`, `index`, or `inScope` are all valid syntactic variables.

Recall the MINI ML grammar in Example 1:

```
toplevel : 'let' VAR '=' expr ';;' toplevel ';;'
         | expr ';;'
```

The `let` top level production is the typical use case for using scoped syntactic variables, since there is a neat correspondence between the semantics of MINI ML `'let'` and Tosca scoped variables. By doing this, the incomplete inline `'let'` (cf. Example 6 and 5) becomes easy to implement. But first, in order to define scoped variables directly in the ANTLR grammar of $\mathcal{L}$, we extended the ANTLR language with three new options: *(i)* `<boundvar=x>`, *(ii)* `<bound=x>`, and *(iii)* `<symbol>`. Option *(i)* can be attached to any terminal symbol, e.g., `VAR<boundvar=x>` to map this terminal symbol onto a bound variable. It also gives an internal id (like `x`) so that it can be referenced in Option *(ii)*, `<bound=x>`, to start a new lexical scope in which the bound variable can occur. Option *(iii)* indicates which terminal needs to be mapped onto a syntactic variable by looking for a bound variable of the same name, in the list of in-scope variables maintained by the parser (cf. Section 4.1). If none is found, then the variable is considered free.

Example 8. Let us rewrite the inline `'let'` example using scoped variables. We start by rewriting the MINI ML ANTLR grammar.

```
toplevel : 'let' VAR<boundvar=x> '=' expr ';;'
                  toplevel<bound=x> ';;' | ...

simple_expr : VAR<symbol> | ...
```

Next we change the corresponding `enum Toplevel`:

```
enum Toplevel | Let( Expr, [Simple_expr] -> Toplevel )
               | Expr( Expr )
```

Finally we replace `VAR` in `enum Simple_expr` by the keyword `allows-variables`:

```
enum Simple_expr | allows-variables | TRUE | FALSE
                 | INT | Expr
```

Now the first `TopLevel` enumeration value `Let` takes two associated values: the let expression bound to `VAR` and the `toplevel` defining the scope of the `let` variable. Notice that the type `Simple_expr` has also been updated to accept syntactic variable by adding `allows-variables` keyword, which means variables are valid values for `Simple_expr` .

Example 9. Finishing the implementation of `InlineLetSimpleExpr` is now trivial:

```
rule InlineLetSimpleExpr(
  toplevel⟦ let #VAR = #simple_expr ;; #toplevel ;; ⟧)
  → #toplevel(#simple_expr)
```

This rule matches the `let` construction binding simple expression, and when the rule is evaluated, it substitutes all occurrences of `#VAR` in `#toplevel` by `#simple_expr`.

On a final note, syntactic variables are not solely limited to encode programming language variables. This mechanism can also be used everywhere there is a need to associate a name to an entity referenced within a context. It includes, but is not limited to, block labels, constants, functions, modules, or packages.

Specificity-Ordered Pattern Matching. Pattern matching is frequently used in compiler generator to accomplish essentially two main tasks: to dispatch on language construct to compile, and to recognize specific program fragments to optimize. Several simple dispatch example were given in Example 2. The next example shows the second case: identifying program fragments that can be optimized.

Example 10. Compiling the MINI ML's `if then else` can be optimized by two more specific rules and one fallback rule.

```
rule CompExpr(expr⟦if true then #expr2 else #expr3⟧)
  → CompExpr(#expr2)

rule CompExpr(expr⟦if false then #expr2 else #expr3⟧)
  → CompExpr(#expr3)

rule CompExpr(expr⟦if #expr1 then #expr2 else #expr3⟧)
  → ...
```

Notice that the optimization is applied on the function `CompExpr` *compiling* Mini ML. It is fairly easy to see that for more complex languages than Mini ML, troubleshooting issues, such as identifying missing or redundant cases, becomes harder. This is especially true when pattern matching is used to single out very specific program fragments which can grow quite large. It is also more difficult to identify when a pattern is more specific than another pattern. The *specificity* compilation phase described in Section 4.3 is about providing a solution to these concerns.

3 From Theory to Practice

In this section we describe the programming language Tosca, starting from its basis in higher-order rewriting and leading to its actual features. In Section 3.1 we describe the theory behind Tosca: higher-order term rewriting, in particular Combinatory Reduction Systems. Section 3.2 gives an overview over the Tosca Core grammar and relates it to CRS. Then Section 3.3 shows how Tosca Core is sweetened for human consumption.

3.1 Combinatory Reduction Systems

Rewriting naturally underlies many, if not all tasks of a standard compiler. Hence it is naturally to encode a compiler generator in a rewriting-based language, an idea which was taken up in crsx [19, 20]. In the case of Tosca the foundational basis is given by *Combinatory Reduction Systems* (*CRSs* for short) [1, 13]. CRSs form a formalism for higher-order rewriting, that is, terms may contain λ-abstractions (aka closures) and bound variables, and we may rewrite such terms.

In contrast to *higher-order rewrite systems* (HRSs for short) [15] which form the formal basis of the interactive theorem prover Isabelle, CRSs are a *second*-order formalism and are untyped. Intuitively that means that a variable can represent a first-order but not a higher-order function. For example, a variable can represent the function plus but not the map function in Example 3. This restriction, however, is not really crucial as HRSs and CRSs are simulation equivalent [22].

In the context of Tosca, CRSs are a more natural formalism, as the rewriting machinary feels less heavy. In particular CRSs can be conceived more naturally as a programming language. In passing we mention that *optimality* has been studied in the context of *Interaction Systems* [2], which form a subclass of CRSs and thus forms a subclass of Tosca Core.

We follow van Raamsdonk [22] in our presentation and restrict to the essentials required in context of this paper. The reader is kindly refered to [22] for further details. We start by introducing the main ingredient: meta terms.

Definition 11. Let $\mathcal{V}$ be a set of *variables*, $\mathcal{M}$ a set of *meta variables*, and $\mathcal{F}$ a set of *function symbols*. Every $X \in \mathcal{M}$ and every $f \in \mathcal{F}$ has a fixed arity. Let $x \in \mathcal{V}$, $X \in \mathcal{M}$, and $f \in \mathcal{F}$. A *meta term* t is

$$t := x \mid f(t_1, \dots, t_n) \mid X(t_1, \dots, t_l) \mid [x]t$$

The expression $[x]t$ provides CRS's notation for λ-abstraction, in particular the variable x is bound in t. A meta term without meta variables is called a *term.*

Recall the formulation of the map function from Example 3. To avoid confusion we make use of a new signature $\mathcal{F} := \{\mathsf{map}, \mathsf{cons}, \mathsf{nil}, \mathsf{plus}, \mathsf{1}\}$. Furthermore, we need the meta variables $\mathcal{M} := \{F, Y_{\mathrm{hd}}, Y_{\mathrm{tl}}, \ldots\}$, and the variables $\mathcal{V} := \{x, \ldots\}$. Then we can construct the meta term: $\mathsf{map}([x]F(x), \mathsf{cons}(Y_{\mathrm{hd}}, Y_{\mathrm{tl}}))$.

Meta terms are considered modulo α-conversion, for example, the meta terms $[x]F(x)$ and $[y]F(y)$ are identified. One may think of bound variables as formal parameters, whose names do not matter too much either. Furthermore, we tacitly assume the variable convention [22].

We define the root $\mathsf{rt}(t)$ of a meta term t according to the grammar given in Definition 11. I.e., $\mathsf{rt}(x) := x$, $\mathsf{rt}(f(t_1, \ldots, t_n)) := f$; $\mathsf{rt}(X(t_1, \ldots, t_l)) := X$, and finally $\mathsf{rt}([x]t) := [x]$.

Similarly the set of *positions* of t is defined as a string of natural numbers to identify occurrences of meta terms in t.

$$\mathsf{Pos}(t) := \begin{cases} \{\varepsilon\} & \text{if } t = x \\ \{\varepsilon\} \cup \{ip \mid p \in \mathsf{Pos}(t_i)\} & \begin{array}{l}\text{if } t = f(t_1, \ldots, t_n)) \\ \text{or } t = X(t_1, \ldots, t_n)\end{array} \\ \{\varepsilon\} \cup \{0p \mid p \in \mathsf{Pos}(t)\} & \text{if } t = [x]t\ . \end{cases}$$

For example with respect to $t := \mathsf{cons}(F(Y_{\mathrm{hd}}), \mathsf{map}([x]F(x), Y_{\mathrm{tl}}))$, the meta term $F(x)$ occurs at position 210. The meta term occurring at position p in t is denoted as $t|_p$.

Based on the definition of meta terms, we want to define rewrite rules. Not every meta term is a good candidate to be the left hand side of a rewrite rule. Fortunately, there is a sensible restriction: *patterns.*

Definition 12. A meta term t is a *pattern*, if for every meta variable $X(t_1, \ldots t_l)$ all arguments t_i are distinct bound variables.

This "pattern restriction" guarantees that unification with a term is decidable. Note that second-order unification in general is undecidable [8]. Thus we can compute whether a "rule matches with term" and if we can "apply a rule". The specifics behind higher-order pattern unification will be given in Section 4.3, where we will present its implementation in Tosca.

To define a rewrite step, we also need the notion of contexts. A context is just that: a context, which is not affected at all by a rewrite step. To formally define contexts, we need a fresh function symbol $\square \notin \mathcal{F}$. A term C with exactly one occurrence of $\square$ is called a *context.* For replacing $\square$ by a term t in C we write $C[t]$. It is important to note, that a context may trigger variables to be bound, i.e., they will be captured. Thus, $[x]f(\square) \neq [y]f(\square)$, as the variable x will be bound in one, but not the other.

Definition 13. A *rule* $\ell \to r$ consists of two meta terms ℓ and r, where *(i)* ℓ is a pattern with root $f \in \mathcal{F}$, *(ii)* all meta variables in r occur in ℓ, and *(iii)* all variables are bound. A set of rewrite rules is a called a *Combinatory Rewrite System.*

By Condition *(i)* the root of the left-hand side has to be a *function symbol.* Condition *(ii)* guarantees that the meta variables r are defined in ℓ, i.e., no new meta variable appears in r. By Condition *(iii)* rules do not contain variables, that are not bound.

Example 14 (continued from Examples 3)**.** The map function can be defined with the following two rewrite rules.

$$\begin{aligned}&\mathsf{map}([x]F(x),\mathsf{nil}) \to \mathsf{nil}\\ &\mathsf{map}([x]F(x),\mathsf{cons}(Y_{\mathrm{hd}},Y_{\mathrm{tl}})) \to \mathsf{cons}(F(Y_{\mathrm{hd}}),\mathsf{map}([x]F(x),Y_{\mathrm{tl}}))\ .\end{aligned}$$

Assuming for example the unary representation of numbers via the symbols plus and 1 and the encoding of lists via cons and nil, we can extend this CRS as we did in Example 4 with a main-function.

$$\mathsf{main} \to \mathsf{map}([x]\mathsf{plus}(1,x),(1,2,3))\ .$$

Now we have defined a rewrite system, i.e., a program. But how to compute something, i.e., how to "apply a rule to a term"? First we think about how we can replace the meta variable F with $[x]\mathsf{plus}(1,x)$. For that we introduce the notion of *substitutions*, *substitutes* and *valuations.*

The *substitution* $s[\vec{x} := \vec{t}]$ of a sequence of terms $\vec{t}$ for a sequence of variables $\vec{x}$, is inductively defined as follows.

$$s[\vec{x} := \vec{t}] := \begin{cases} t_i & \text{if } s = x_i \in \vec{x}\\ y & \text{if } s = y \notin \vec{x}\\ f(s_1[\vec{x} := \vec{t}],\dots,s_l[\vec{x} := \vec{t}]) & \text{if } s = f(s_1,\dots,s_l)\\ [y]s'[\vec{x} := \vec{t}] & \text{if } s = [y]s' \wedge y \notin \vec{x}\end{cases}$$

For the last case it is important, that no variable becomes bound, thus $y \notin \vec{x}$. However, this is easily assumed making tacit use of α-conversion.

Let s be a term, and $x_1,\dots,x_n$ be variables. The *n-ary substitute*, denoted as $\underline{\lambda} x_1 \dots x_n.\, s$, is defined as follows:

$$(\underline{\lambda} x_1 \dots x_n.\, s)\vec{t} := s[\vec{x} := \vec{t}]\ .$$

Let σ be a mapping that assign n-ary substitutes to n-ary meta variables and let t be a meta term. A *valuation* is the homomorphic extension of σ, inductively defined as follows.

$$t^\sigma := \begin{cases} x & \text{if } t = x\\ f(t_1^\sigma,\dots t_l^\sigma) & \text{if } t = f(t_1,\dots,t_l)\\ [x]t'^\sigma & \text{if } t = [x]t'\\ \sigma(X)(t_1^\sigma,\dots,t_k^\sigma) & \text{if } t = X(t_1^\sigma,\dots,t_k^\sigma)\end{cases}$$

Applying the above definition is straight forward. In particular $\sigma(X)$ means replace X by its image.

We illustrate the interplay of substitutions, substitutes and valuations by an example.

Example 15. For $\sigma = \{F \mapsto \underline{\lambda} x.\, \mathsf{plus}(1, x), Y_{\text{hd}} \mapsto 1, Y_{\text{tl}} \mapsto \mathsf{nil}\}$ we get

$$\begin{aligned}
&(\mathsf{map}([x]F(x), \mathsf{cons} Y_{\text{hd}}, Y_{\text{tl}})))^{\sigma}\\
&= \mathsf{map}([x](\underline{\lambda} x.\, \mathsf{plus}(1, x))(x), \mathsf{cons}(1, \mathsf{nil}))\\
&= \mathsf{map}([x]\mathsf{plus}(1, x), \mathsf{cons}(1, \mathsf{nil}))\ .
\end{aligned}$$

Similarly, we get

$$\begin{aligned}
&(\mathsf{cons}(F(Y_{\text{hd}}), \mathsf{map}([u]F(u), Y_{\text{tl}}))^{\sigma}\\
&= \mathsf{cons}((\underline{\lambda} x.\, \mathsf{plus}(1, x))(1), \mathsf{map}([x](\underline{\lambda} x.\, \mathsf{plus}(1, x))(x), \mathsf{nil}))\\
&= \mathsf{cons}(\mathsf{plus}(1, 1), \mathsf{map}([x]\mathsf{plus}(1, x), \mathsf{nil}))
\end{aligned}$$

We emphasize that CRSs make use of λ-abstraction on two levels. One the one hand one employs binders on the object level, where abstractions are denoted as $[x]t$. On the other hand, binders are used on the meta level in the form of substitutes, denoted as $\underline{\lambda} x.\, s$. Furthermore we note that the interplay between substitutions, substitutes and valuations in CRSs amounts to performing a complete β-development. Finally, we can give a declarative definition of a rewrite step.

Definition 16. A term s *rewrites* to a term t, denoted by $s \to t$, if $s = C[\ell^{\sigma}]$ and $t = C[r^{\sigma}]$ for a context C, a valuation σ, and a rule $\ell \to r$.

Example 17. We have the rewrite step

$$\begin{aligned}
&\mathsf{map}([u]\mathsf{square}(u), \mathsf{cons}(1, \mathsf{nil})) \to\\
&\quad \mathsf{cons}(\mathsf{square}(1), \mathsf{map}([u]\mathsf{square}(u), \mathsf{nil}))
\end{aligned}$$

with $C = \square$, σ from Example 15, and the second rewrite rule from Example 14.

Revisiting Example 14 and taking stock of the structure of the rules, we see that the provided CRS is restricted in three crucial points:

1. Each meta variable on the left-hand side occurs at most once.
2. None of the arguments of the left-hand side contains symbols *defined* by any of the other rules, that is, neither nil nor cons occur as root of a left-hand side.
3. The rules are not *overlapping*, that is, the rules are non-ambigious.

These observations motivate the next definitions. Let a CRS $\mathcal{R}$ over the signature $\mathcal{F}$ be given. Then $\mathcal{R}$ is called *left-linear*, if every meta variable on the left-hand side of a rule occurs at most once. Further, we can write the signature $\mathcal{F}$ as the disjoint set of symbols $\mathcal{D}$ and $\mathcal{C}$, where $\mathcal{C}$ collects all symbols which do *not* occur as root symbol of a left-hand side in $\mathcal{R}$. If all arguments of left-hand side contain only symbols from $\mathcal{C}$, then $\mathcal{R}$ is called a *constructor CRS*.

Let s and t be meta terms. We say that s *overlaps* with t, if there exists a valuation σ and a non-meta variable position p in s such that $s|_p{}^{\sigma} = t^{\sigma}$. Two rules $l_1 \to r_1$ and

$l_2 \to r_2$ are *overlapping* if l_1 overlaps with l_2 or vice versa and the overlap does not occur at the root when two copies of the same rule are considered. Finally, if there are no overlapping rules, then $\mathcal{R}$ is called *non-overlapping*. Left-linear and non-overlapping CRSs, are called *orthogonal*.

If any successful evaluation yields the same result, i.e., we do not have to worry about non-determinism, we call a rewrite system *confluent*, or *Church-Rosser*. We will make use of the following result later on, cf. Section 4.3.

Theorem 18. *[13] Orthogonal Combinatory Reduction Systems are* confluent*.*

As mentioned above, *Interaction systems*, introduced by Asperti et al. [3, 4] form a subset of CRSs. More precisely interaction systems form a subset of constructor, orthogonal CRSs and are morally equivalent.

3.2 Tosca Core

In this section we describe the programming language Tosca, focusing on its foundation in higher-order rewriting. The Tosca Core is a subset of the Tosca language. The idea behind Tosca Core is to reduce the rich programming language Tosca to its bare necessities, which is in direct correspondence to left-linear, constructor CRSs defined in the last section.

Having a small core makes it easier to reason about it, easier to process, and easier to analyze automatically. So what is a necessity? We distinguish between semantic and technical necessities. Semantic relates to the expressibility of Tosca, technical to the implementation.

The ANTLR grammar of the core is available at `Core.g4`. As in Section 3.1 we start with the definition of meta terms or, as in Tosca Core: `cterms`. Here, the `c` stands for "core".

```
cterm : VARIABLE
      | CONSTRUCTOR cterms?
      | METAVAR cterms?
      | '[' VARIABLE ']' cterm
      | literal

cterms : '(' cterm (COMMA cterm)* ')'
```

A `cterm` looks similar to a meta term in Definition 11. The first four cases directly relate. A `VARIABLE` corresponds to an element in $\mathcal{V}$, and starts with a lower case letter, e.g., `x`. Similarly, a `CONSTRUCTOR` corresponds to a function symbol in $\mathcal{F}$.[6] In Tosca function symbols start with upper case letters, e.g., `F`. Next, `METAVAR` indicates elements in $\mathcal{M}$, which start with `#`, e.g. `#X`. Finally, the case in line 4 binds the `VARIABLE` in the `cterm`, e.g., `[x]F(x)` corresponds to $[x]f(x)$. The next case is a `literal`:

```
literal : STRING
        | NUMBER
```

[6]We emphasize that the keyword `CONSTRUCTOR` means function symbol in the signature, rather than merely constructor as implicit in the above notion of *constructor* CRSs.

Literals are built-in in Tosca—purely for efficiency reasons, i.e., it is a technical necessity. They are conceptually not difficult and easily expressible in CRSs.

The arguments of `CONSTRUCTOR` and `METAVAR` are *optional* `cterms`. This is the implementation of "dropping the parenthesis" for function symbols without arguments. That is, we usually write simply `True` and not `True()`—which we would need to write, strictly according to Definition 11.

The main concepts of CRS in Section 3.1—patterns, meta terms, contexts—are directly transferable to `cterms`.

The `cterms` are the basic building block of a Tosca program. Now we see, how to create a Tosca Core program, that is, a rewrite system.

```
ctransscript : cdecl+

cdecl : 'rule' cterm '→ ' cterm
        | 'enum' CONSTRUCTOR
        | ...
```

Every Tosca Core program consists of a sequence of declarations, namely `cdecl`. One case `cdecl` is a rule declaration, also indicated by the `rule` keyword. Similarly to Section 3.1 above, a rule consists of two terms, separated by `'→ '`. As in Definition 13 restrictions need to be imposed. However, due to practical concerns some lesser constraints have been weakened. For example the following Tosca rule `F(x) → x` is valid in Tosca Core, but not as CRS. The variable x is free, while Definition 13(iii) demands that x is bound. This is a practical necessity. We need free variables to be able to handle invalid embedded programs in which some variable are used without declaring them.

But in `cdecl`, what is hidden behind ... ? For one, we have import declarations. This technicality is needed to generate Java code, but not the focus of this section. Also hidden behind ... are type declarations, whose full incorporation is work in progress. We briefly report on this when we mention future work in Section 6. One subtle, but severe difference is that in a Tosca program, the rules are given in a *sequence* ie., in lexical order. In the CRS setting, Definition 13, a program corresponds to a *rewrite system*, which is a *set* of rules and does not enforce any order on the rules. We will investigate this difference, and the consequences, in more detail in Section 4.3.

3.3 Tosca Core To Tosca

The previous sections built enough of the necessary scaffolding in order to demonstrate Tosca is based on solid CRS foundation. The last stage consists of removing the constraint on bare necessities to add *syntactic sugar* to makeTosca programs more consumable.

To show how Tosca Core is expanded to Tosca, we first need to formalize the subset of the Tosca syntax introduced in the Section 2.2. The syntax for defining types has been left out since the current implementation mostly ignores types. The complete grammar is available at `Tosca.g4`.

```
transscript    : decl+

decl           : 'rule' CONSTRUCTOR args? → terms
```

$$
\begin{array}{lcl}
\mathcal{C}_{transcript}\llbracket\ decl_1\ \ldots\ decl_n\ \rrbracket & = & \mathcal{C}_{decl}\llbracket\ decl_1\ \rrbracket \ldots \mathcal{C}_{decl}\llbracket\ decl_n\ \rrbracket \\
\mathcal{C}_{decl}\llbracket\ \texttt{rule CONSTRUCTOR}\ args?\ \rightarrow\ terms\ \rrbracket & = & \texttt{rule CONSTRUCTOR}\ \mathcal{C}_{args}\llbracket\ args?\ \rrbracket \rightarrow \mathcal{C}_{term}\llbracket\ terms\rrbracket \\
\mathcal{C}_{decl}\llbracket\ \texttt{enum CONSTRUCTOR}\ \rrbracket & = & \texttt{enum CONSTRUCTOR} \\
\mathcal{C}_{term}\llbracket\ \texttt{CONSTRUCTOR}\ args?\ \rrbracket & = & \texttt{CONSTRUCTOR}\ \mathcal{C}_{args}\llbracket\ args?\ \rrbracket \\
\mathcal{C}_{term}\llbracket\ \texttt{()}\ \rrbracket & = & \texttt{Nil} \\
\mathcal{C}_{term}\llbracket\ \texttt{(}\ term\ \texttt{)}\ \rrbracket & = & \mathcal{C}_{term}\llbracket\ term\ \rrbracket \\
\mathcal{C}_{term}\llbracket\ \texttt{(}\ term_1\ \texttt{,}\ term_2\ \ldots\ \texttt{,}\ term_n\ \texttt{)}\ \rrbracket & = & \texttt{Cons(}\ \mathcal{C}_{term}\llbracket\ term_1\ \rrbracket\ \texttt{,}\ \mathcal{C}_{term}\llbracket\ term_2\ \ldots\ term_n\ \rrbracket\ \texttt{)} \\
\mathcal{C}_{term}\llbracket\ \texttt{VARIABLE}\ \rrbracket & = & \texttt{VARIABLE} \\
\mathcal{C}_{term}\llbracket\ \texttt{STRING}\ \rrbracket & = & \texttt{STRING} \\
\mathcal{C}_{term}\llbracket\ \texttt{NUMBER}\ \rrbracket & = & \texttt{NUMBER} \\
\mathcal{C}_{term}\llbracket\ \texttt{METAVAR}\ margs?\ \rrbracket & = & \texttt{METAVAR}\ \mathcal{C}_{margs}\llbracket\ margs?\ \rrbracket \\
\mathcal{C}_{term}\llbracket\ concrete\ \rrbracket & = & \mathcal{C}_{concrete}\llbracket\ concrete\ \rrbracket \\
\mathcal{C}_{args}\llbracket\ \texttt{()}\rrbracket & = & \texttt{()} \\
\mathcal{C}_{args}\llbracket\ \texttt{(}\ scope_1\ \ldots\ \texttt{,}\ scope_n\ \texttt{)}\rrbracket & = & \texttt{(}\ \mathcal{C}_{scope}\llbracket\ scope_1\rrbracket\ \ldots\ \texttt{,}\ \mathcal{C}_{scope}\llbracket\ scope_n\rrbracket\ \texttt{)} \\
\mathcal{C}_{scope}\llbracket\ \texttt{[}\ binders\ \rrbracket & = & \mathcal{C}_{binders}\llbracket\ binders\ \rrbracket \\
\mathcal{C}_{scope}\llbracket\ term\ \rrbracket & = & \mathcal{C}_{term}\llbracket\ term\ \rrbracket \\
\mathcal{C}_{binders}\llbracket\ \texttt{VARIABLE}\text{<boundvar=x>}\ binders\text{<bound=x>}\ \rrbracket & = & \texttt{[VARIABLE}\text{<boundvar=x>}\ \texttt{]}\ \mathcal{C}_{binders}\llbracket\ binders\ \rrbracket\text{<bound=x>} \\
\mathcal{C}_{binders}\llbracket\ \texttt{]}\ term\ \rrbracket & = & \mathcal{C}_{term}\llbracket\ term\ \rrbracket \\
\mathcal{C}_{margs}\llbracket\ \texttt{()}\rrbracket & = & \texttt{()} \\
\mathcal{C}_{margs}\llbracket\ \texttt{(}\ term_1\ \ldots\ \texttt{,}\ term_n\ \texttt{)}\rrbracket & = & \texttt{(}\ \mathcal{C}_{term}\llbracket\ term_1\rrbracket\ \ldots\ \texttt{,}\ \mathcal{C}_{term}\llbracket\ term_n\rrbracket\ \texttt{)}
\end{array}
$$

Figure 2: Expanding Tosca to Tosca Core.

```
               | 'enum' CONSTRUCTOR

term          : CONSTRUCTOR args?
               | '(' (term (',' term)*)? ')'
               | VARIABLE
               | STRING
               | NUMBER
               | METAVAR margs?
               | concrete

args           : '(' (scope (',' scope)*)? ')'
scope          : '[' binders
               | term
binders        : VARIABLE<boundvar=x> binders<bound=x>
               | ']' term
margs          : '(' (term (',' term)*)? ')'
```

The equations formalizing the correspondence between Tosca and Tosca Core are shown in Figure 2. Here, $\mathcal{C}$ stands for $\mathcal{C}$orify. The subscripts indicate the ANTLR production in question. To improve readability, Tosca constructs are presented on the equation left-hand side. It is easy to see that all cases from the above grammar are fully covered. Most of the equations are straightforward and self-explanatory. The list syntax is expanded to nested `Cons` and `Nil` values. The main difficulty is normalizing embedded programs, done by $\mathcal{C}_{concrete}$; Section 4.2 presents this process in details.

4 Compilation

Tosca is a self-hosted, source-to-source compiler generator. Its goal is to translate Tosca programs to feature rich programming languages such as Java, C++ or Haskell—where the current implementation genearates Java 8 programs. In this section, we show the internals of Tosca. Tosca works as a preprocessor. In other words: it does not rely on a full compilation stack. As such, the Tosca architecture is very simple, as depicted in Figure 3. This is because only features, and associated lowerings [6] as well as optimizations, present in Tosca but not in the target language need non-trivial translation.

To give an example: function closure, a common feature found in many functional programming languages, including Tosca, is usually translated into a lower-level language using closure conversion [11]. However since Java supports closure—called anonymous class, or more recently *lambda expressions* [9]—Tosca does not need to apply this technique when translating to Java.

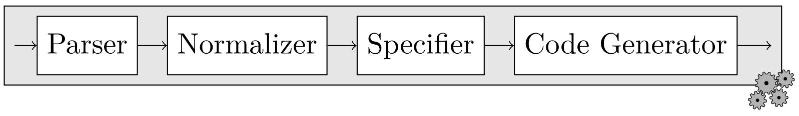

Figure 3: The Tosca Architecture.

Going back to Figure 1, Figure 3 shows what is happening inside the Tosca compiler generator. The Tosca compilation phases—the Parser, Normalizer, Specifier, and Code Generator—are described in details below.

Each compilation phase takes a (human) readable input and produces a (human) readable output. Of course, some intermediate steps are easier to read than others, but we took a lot of effort to remain readable throughout. In particular, we find this important for the Java source code—i.e., the final output. This approach is very advantageous for debugging and unit testing each compilation phase individually. This is a common approach found in other compilers; Tosca systematizes it.

4.1 Parser

The parser reads Tosca program and produces an AST of the Tosca program. This AST contains unparsed embedded program fragments, which are parsed during the normalization phase (cf. Section 4.2).

$$
\begin{array}{lcl}
\mathcal{T}_{grammar}\llbracket\ rule_1 \ \ldots\ rule_n\ \rrbracket & = & \mathcal{T}_{rule}\llbracket\ rule_1\ \rrbracket \ \ldots\ \mathcal{T}_{rule}\llbracket\ rule_n\ \rrbracket \\
\mathcal{T}_{rule}\llbracket\ \texttt{NAME}\ :\ alt\ \rrbracket & = & \texttt{enum NAME(}\ \mathcal{T}_{alt}\llbracket\ alt\ \rrbracket_{NAME}\ \texttt{)} \\
\mathcal{T}_{rule}\llbracket\ \texttt{NAME}\ :\ alt_1\ \ldots\ alt_n\ \rrbracket & = & \texttt{enum NAME}\ \mathcal{T}_{alt}\llbracket\ alt_1\ \rrbracket_{NAME_1}\ \ldots\ \mathcal{T}_{alt}\llbracket\ alt_n \\
 & & \rrbracket_{NAME_n} \\
\mathcal{T}_{alt}\llbracket\ elmt_1\ \ldots\ elmt_n\ \rrbracket_{Name} & = & \texttt{| Name(}\ \mathcal{T}_{elmt}\llbracket elmt_1\rrbracket\ \ldots\ \mathcal{T}_{elmt}\llbracket\ elmt_n\ \rrbracket\ \texttt{)} \\
\mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ \rrbracket & = & \texttt{NAME} \\
\mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \rrbracket & = & \texttt{List(NAME)} \\
\mathcal{T}_{elmt}\llbracket\ (\ elmt_1\ \ldots\ elmt_n\ )\ \rrbracket & = & \mathcal{T}_{elmt}\llbracket\ elmt_1\ \rrbracket\ \ldots\ \mathcal{T}_{elmt}\llbracket\ elmt_n\ \rrbracket \\
\mathcal{T}_{elmt}\llbracket\ (\ elmt_1\ \ldots\ elmt_n\ )\ ebnf\ \rrbracket & = & \mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \rrbracket\ \&\ \mathcal{T}_{rule}\llbracket\ \texttt{NAME}\ :\ elmt_1 \\
 & & \ldots\ elmt_n\ \rrbracket \\
\mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \texttt{<bound=VARNAME>}\ \rrbracket & = & \mathcal{E} \\
\mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \texttt{<boundvar=VARNAME>}\ \rrbracket & = & [\ \texttt{NAME}\ ]\ \texttt{->}\ \mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \rrbracket \\
\mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \texttt{<symbol>}\ \rrbracket & = & \mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \rrbracket \\
\mathcal{T}_{elmt}\llbracket\ \texttt{NAME}\ ebnf\ \texttt{<sugar>}\ \rrbracket & = & \mathcal{E}
\end{array}
$$

Figure 4: Translating from the ANTLR grammar to Tosca types

The current Tosca implementation relies on the parser generator ANTLR v4 [17] to represent both the Tosca grammar and the embedded program grammars to be processed by Tosca. The choice of ANTLR v4 has mostly been driven by the following characteristics: It has familiar EBNF-like notation making ANTLR grammars easy to write and read. Furthermore it has great support for automatic direct left factoring and associativity, as well as a great performance due to adaptive LL(*) parsing [18]. Finally, it supports multiple target languages (e.g., JAVA, JavaScript) and it generates readable code, which is one of the priorities of Tosca. All this fits very well in the Tosca philosophy of ease-of-use, without compromising functionality and performance.

Tosca is able to handle programs written in different languages by defining common *mapping rules* from ANTLR grammar concepts to Tosca typed expression. These mapping rules are fundamental for implementing several tools provided by Tosca including the *(i)* Tosca parser, *(ii)* parsers for the embedded program fragments, and also *(iii)* a Tosca type description corresponding to Tosca expressions recognized by these parsers. Figure 4 shows these mapping rules for the following simplified grammar of ANTLR itself:

```
grammar    : rule*
rule       : NAME : alt*
alt        : elmt*
elmt       : NAME ebnf? option? | block ebnf?
block      : ( alt* )
ebnf       : '?' | '*' | '+'
option     : <sugar> | <bound=VARNAME>
           | <boundvar=VARNAME> | <symbol>
```

As seen before, an ANTLR grammar consists of a set of *grammar rules*. Each rule is identified by an unique `NAME`, and defines a list of *alternatives* (`alt`). Each alternative is defined as sequence of *elements* (`elmt`) composing it, which in turn corresponds to *token* of a given `NAME` and `block`.

The mapping of ANTLR grammar onto Tosca is shown in Figure 4. The mapping rules are of the form

$$\mathcal{T}_{rulename}[\![\text{ANTLR }]\!] = \texttt{Tosca type},$$

where $\mathcal{T}_{rulename}$ ($\mathcal{T}$ for $\mathcal{T}$*ranslation*) is a function on an ANTLR construct, *rulename* is one of the ANTLR grammar rule names given above and `NAME` an environment containing the current alternative name (see below).

Each grammar *rule* is translated into a Tosca enumeration of the corresponding rule name. For a grammar rule with only a single alternative, the alternative name is set to the name of the rule. Otherwise the alternative name is set to the rule name suffixed by the alternative position in the rule. A rule reference becomes an enumeration value. Repetition, either *zero or one element* (`'?'`), *zero or more elements* (`'*'`) or *one or more elements* (`'+'`), maps to list of values. While this mapping is approximate, it has the advantage of being compatible with the Tosca standard library operating on lists (cf. Section 5). A block with repetition cannot be mapped directly to a Tosca enumeration. Tosca eliminates blocks by lifting the block elements to a top level grammar rule (The 3rd and 4th rule of $\mathcal{T}_{elmt}$). The & symbol indicates an additional type is created. Elements with the option `<sugar>` are filtered out during parsing and therefore do not appear in Tosca (as represented by $\mathcal{E}$).

The parser turns ANTLR events into Tosca expressions following the type mapping rules. The only difficulty comes from handling scoped variables. It works as follows: custom actions are added to the ANTLR grammar to indicate which event corresponds to either a bound variable (`<boundvar>`), the beginning and the end of a scope (`<bound>`), or to a variable occurrence (`<symbol>`). These action allows the parser to internally keep track of in-scope variables needed to create bindings upon variable occurrences. If binding creation fails, i.e., no variable is found in the in-scope variables, a fresh variable is created.

4.2 Normalizer

The normalizer turns the AST with embedded program fragments and syntactic sugar, as produced by the parser, into Tosca Core. The formal description is presented in Section 3.3—here we give the implementation details.

The main task performed by the normalizer is to expand embedded program fragments into Tosca and then Tosca Core, as shown in Example 5 and Example 6 in Section 2.2.

A further task is to translate syntactic sugar constructs into their core representation equivalent. In both tasks it is important to preserve the original program location to produce high-fidelity error messages by showing Tosca program snippets written by programmers.

The normalizer relies on *meta parsers* to parse the embedded programs, which are derived from *meta grammars.*

Definition 19. A *meta grammar* is a grammar that has been augmented with additional rules to be able to: *(i)* embed Tosca expressions in places where terminals and non-terminals occur, *(ii)* directly declare meta variables in the rule's left-hand side and reference them in the rule's right-hand side, and *(iii)* parse program fragments, delimited by a grammar rule.

Tosca is capable of automatically generating meta grammars from a given grammar. A snippet of the generated grammar for MINI ML is shown in Example 20—and for the bootstrapping we also generated a meta grammar for the Tosca language itself (cf. Section 5). This is accomplished as follows:

- Each grammar rule corresponds to two meta grammar rules: the first one matches the original grammar rule with some extensions, and the second one, which by convention is suffixed by `_EOF`, allows the grammar rule to represent the whole program.
- Two alternatives are added to each grammar rule: one to parse meta variables (in the meta grammar), and the other to escape back to the Tosca program. By default, meta variable tokens are prefixed by the character `#`. In case of conflict with the target language, another character can be chosen.
- Embedded Tosca programs are read as a single stream of characters (a token), which is fed to the Tosca parser to produce the AST. The AST is recursively normalized, which in turn can trigger subsequent parsing and normalization translations. This recursive process stops when the AST contains no more embedded code.

Example 20. The meta grammar corresponding to the `toplevel` and `expr` grammar rules from Example 1 is given next:

```
toplevel_EOF : toplevel EOF

toplevel : 'let' var_TOK '=' expr ';' toplevel ';;'
         |  expr  ';;'
         |  '#toplevel'
         | ET_toplevel

var_TOK_EOF : var_TOK EOF

var_TOK : VAR | '#VAR'
         | ET_var_TOK

expr_EOF : expr EOF

expr : timesExpr '+' expr
     | timesExpr '-' expr
     | timesExpr
     | '#expr'
     | ET_expr

// Lexer rules

```

```
ET_toplevel: '⟨toplevel:' -> pushMode(Embed);
ET_var_TOK : '⟨VAR:' -> pushMode(Embed);
ET_expr: '⟨expr:' -> pushMode(Embed);

mode Embed ;
EMBED_END  : '⟩' -> popMode;
EMBED_NESTED: '⟨' -> pushMode(NestedEmbed), more;
EMBEDDED   : . -> more;

mode NestedEmbed;
NESTED_EMBED_END : '⟩' -> popMode, more;
NESTED_EMBED_NESTED: '⟨' -> pushMode(NestedEmbed), more;
NESTED_EMBEDDED  : . -> more;
```

This example clearly shows the `_EOF` grammar rules, as well as the additional rule alternatives explained above. Parsing embedded Tosca code is done by defining two special lexer modes, `Embed` and `NestedEmbed` allowing nesting.

After normalization is done, the AST contains only Tosca Core and is ready for the specificity phase.

4.3 Specifier

A very reasonable expectation of a programming language is that it is deterministic. But term rewriting stems from equational reasoning—and not programming. Thus, as opposed to (functional) programs, term rewrite systems are not deterministic *per se*.

Example 21. We want to implement integer division and start with the following three rules.

```
rule Div(0, #Y) → 0
rule Div(#X, 0) → NaN
rule Div(#X, #Y) → Plus(1, Div(Minus(#X, #Y), #Y))
```

But if we now consider the term `Div(0,0)`, three different rewrite steps are possible:

$$\begin{tikzcd}
 & \texttt{Div(0,0)} \arrow[dl] \arrow[d] \arrow[dr] & \\
\texttt{0} & \texttt{NaN} & \texttt{Plus(Div(Minus(0,0),0))}
\end{tikzcd}$$

Now the task of the *specifier* is to resolve such an ambiguous situation based on specificity criteria. While this phase is not strictly needed in the basic Tosca architecture, it is a unique feature rarely found in other programming languages and therefore presented here—but not yet fully integrated in the compilation phases. So why does this problem occur in the first place? Because the three rules *overlap* (cf. Section 3.1). In particular, we have an overlap between the first two rules with `#X` $\mapsto$ `0` and `#Y` $\mapsto$ `0` and the second and third rule with `#Y` $\mapsto$ `0`.

So how to disambiguate this situation? In Example 21 we may observe that there are different kinds of overlaps. If we look at the overlap between the second and third rule, we see that one is more *specific* than the other. That is, `Div(#X, 0)` is an instance of `Div(#X,#Y)`. It is reasonable to assume that the third rule is conceived as the "fallback" rule by the programmer.

But between the first and the second rule we have a completely ambiguous situation—and in fact, we chose to inform the programmer of this ambiguity, i.e., we raise an exception.

Following [12], we take this as a basis for disambiguation through specificity: We build a *specificity tree.*

Example 22. For the following four terms

```
Div(#X, #Y), Div(0, #Y), Div(0, 0), Div(NaN, #Y)
```

we can build the specificity tree, where the corresponding σs are given on the edges.

$$\begin{tikzcd}
& \texttt{Div(\#X, \#Y)} \arrow[dl, "\texttt{\#X} \mapsto \texttt{0}"'] \arrow[dr, "\texttt{\#X} \mapsto \texttt{NaN}"] & \\
\texttt{Div(0, \#Y)} \arrow[d, "\texttt{\#Y} \mapsto \texttt{0}"'] & & \texttt{Div(NaN, \#Y)} \\
\texttt{Div(0,0)} & &
\end{tikzcd}$$

But if we additionally add `Div(#X, 0)`, we have a parallel overlap, i.e., a situation which cannot be handled by specificity.

By applying the rule in post order with respect to the specificity tree, we try the most specific rule first. That is, we have the rules ordered by specificity—or an error if it is non-solvable: a parallel overlap. Compared to, e.g., [14], we may relate the following concepts: a set of rewrite rules is completely defined iff it is exhaustive. A pattern is useless, if there is a (non-parallel) overlap.

The specificity tree, as presented, can serve for several analyses and transformations. For example, we can use the tree to figure out whether a function symbol is completely defined. Or, we may transform the rules by instantiating less specific rules and thereby remove overlaps—an approach currently taken by CRSX [20].

In particular the following result helps for disambiguation. Recall, that Tosca rules are left-linear (cf. Section 3.2). Thus an application of Theorem 18 yields confluence, as soon as we can guarantee that the rules are non-overlapping.

So far, we presented why we need the specifier and the potentials of the specificity tree. But how do we actually find overlaps? To detect them, we implement *higher-order unification for patterns.* This implementation is based on [16, Figure 2], and presented here briefly. The following example is suitably adapted from Nipkow [16].

Example 23. As an illustrative example we show how to compute σ between the two concrete terms: `F(#X, [x y] -> #F(x))` and `F(0, [x y] -> C(#G(y x)))`. So we give intermediate steps in Tosca syntax (or multiple steps denoted by $\to^*$). The function `Unify` takes a `Pair` of terms and a σ, which is empty (`{}`) at the beginning.

```
Unify(Pair(F(#X, [x y] -> #F(x)),
           F(0, [x y] -> C(#G(y x)))), {})
→* If(ConstructorEqual(F,F),
  Foldl([args sigma] -> Unify(args, sigma), {},
        (Pair(#X, 0),
         Pair([x y] -> #F(x), [x y] -> C(#G(y,x)))
```

To `Unify` the two terms we first compare their root symbols. As they are both constructors and equal, we proceed to `Foldl` over the arguments. To unify the first pair of arguments is straightforward.

```
rule Unify(Pair(#X, 0), {}) →* { #X ↦ 0 }
```

To unify those two terms, we have to map the meta variable `#X` to the term `0`. The next pair of arguments is more involved as it contains functions.

```
Unify(Pair([x y] -> #F(x)), [x y] -> C(#G(y, x))), {})
→* Unify(Pair(#F(x), C(#G(y,x))), {})
```

We can strip away the variables `x y`, as they are equal. In general, we have to deal here with implementing α-conversion.

```
→* Unify(Pair(#H(x), #G(y,x)), {#F ↦ [x] -> C(#H(x))})
```

In this step, we have to introduce a fresh meta variable `#H`. We map the meta variable `#F` to a function, which takes one argument `x` and starts with the function symbol `C`, which has a argument our fresh meta variable `#H` with one argument—as `#F`. For the last step, i.e., unifying `Pair(#H(x), #G(y,x))`, we have to identify the two meta variables `#H` and `#G`. Therefore, we map them both to a fresh meta variable `#H2`.

```
→* { #H ↦ [x] -> #H2(x), #G ↦ [y x] -> #H2(x),
     #F ↦ [x] -> C(#H2(x))}
```

We have to make sure, to also apply `#H ↦ [x] -> #H2(x)` to the image of `#F`.

4.4 Code Generator

Finally the last compilation phase is the code generator producing human readable JAVA source code. The generated code relies on a common runtime JAVA library providing the following capabilities: *(i)* A generic way to represent terms. Tosca functions and enumerations share the same generic term representation, with the only difference that functions are associated with a generated JAVA method corresponding to the function rules. *(ii)* A substitution algorithm which traverses the term tree to look for bound variable occurrences, and substitutes them by other terms. *(iii)* A top level normalization algorithm that repeatedly attempts to reduce the outermost functional term until there are none, or the reduction is stuck. *(iv)* A way to pretty print terms to a generic term format. *(v)* A way to construct terms.

As mentioned in Capability *(i)* for each Tosca function corresponds to a single JAVA method that has the following structure:

- Each Tosca function's rule corresponds a JAVA labeled block—in the same lexical order.

- The first part of the JAVA labeled block encodes the rule's left hand side. It checks for construction symbols and associates meta variables to terms. If the pattern matching fails, a break command is issued to jump to the next JAVA block.
- The rule contraction is generated in the second part of the JAVA labeled block. A new term is generated by using a combination of new term construction, associated meta variables, and substitution.

Example 24. The JAVA code corresponding to MiniML `EvalExpr` is the following one:

```
1  static Boolean evalExpr(Sink sink, Term term) {
2    if (sink.context().sd++ < 256)  {
3       rule1 : {
4         term = force(sink.context(), term);
5         if (term.descriptor() != MiniML_xexpr_xA1)
6           break rule1;
7         Term sub0 = term.sub(0).ref();
8         /* #timesExpr=sub0 */
9         Term sub1 = term.sub(1).ref();
10        /* #expr=sub1 */
11        sink.start(Plus);
12        sink.start(EvalTimesExpr);
13        sink.copy(sub0.ref());
14        sink.end();
15        sink.start(EvalExpr);
16        sink.copy(sub1.ref());
17        sink.end();
18        sink.end();
19        return true;
20      }
21      rule2 : { ... }
22      rule3 : { ... }
23    }
24     return thunk(sink, evalExpr, term);
25   }
```

The most important characteristics of the JAVA code in Example 24 are the following ones: The test in line 2 makes sure the stack does not overflow. It relies on a common known technique called *trampoline*. When the stack depth reaches an arbitrary threshold (there is no way to know the JAVA max stack depth), the method evaluation is interrupted and a corresponding thunk is returned in line 24. Here, the Boolean value indicates, whether we have a failed or a successful step. The first rule starts at line 3, the second rule and third are sketched in line 21 and 22. Then, `force` evaluates a term enough to be in head normal form—as can be seen in line 4. Line 5 corresponds to the beginning of pattern matching and line 7 is the beginning of the right-hand side rules.

5 Using Tosca—For Real

Tosca defines many standard libraries. Most of the functions in the standard libraries have been written in Tosca itself. However in the spirit of Tosca being light-weight, severals have been written directly in Java. At the Tosca level, only the function signature is specified along with the keyword `extern` to indicate the actual implementation is done in Java. We describe the standard library here briefly:

- Core defines basic functionally `Eval` to force evaluation, `Error`, and `Trace` to trace computation, as well as `Option` and `Boolean`.
- Num defines functions on (and external interfaces to) on numbers like `Plus`, `Minus` . . .
- List gives list functionalities as expected: `Map` (as described in Section 2.2), `Foldl`, `Foldr`, `Filter`, . . . but also, of course, `Head`, `Tail`, `Append` . . .
- Map defines the interfaces for manipulating hash map, such as `MapPut` to add a key-value pair, or `MapKeys` to get the list of keys in a map.
- Pair defines pairs and and functionality on them.
- String defines basic functionalities on Strings such as `UpCase`, `ConcatString`, and interfaces for external string functions.

To ease development, we have developed a Tosca package for the Atom text editor [7]. It provides the basis for syntax highlighting. It is available here: Tosca mode.

Tosca is a very young system. Still it has been used successfully to write a Tosca compiler in Tosca itself. Indeed since Tosca is designed to generate compilers, it is natural to use it for itself. While several existing programming languages are self-hosted (e.g., Lisp or Rust), self-hosting Tosca adds some extra twists. Thus we want to take the remainder of this section to describe our experiences with using and bootstrapping Tosca.

The following steps describe the bootstrapping procedure. The first step has been to write a minimal Tosca compiler using a subset of the crsx language. It only makes use of few core features (simple pattern matching, no overlapping rules) available in both crsx and Tosca. This allowed us to have a first compiler $\mathcal{C}1$ taking Tosca program and generating Java code. This compiler still depends on crsx. The second step was to port the crsx code to Tosca, which was very mechanical due to the high-overlap between the two languages. The third step was to generate a second stand alone compiler $\mathcal{C}2$ using $\mathcal{C}1$ and crsx. Finally, self-hosting is truly achieved by generating a third compiler $\mathcal{C}3$ using $\mathcal{C}2$.

One twist came from the use of embedded syntax—because in our case this was now Tosca syntax. The main issue is to have multiple versions of the same language, Tosca in our case, coexist in the same Java Virtual Machine (JVM). For instance, consider this actual source code snippet, which defines a Tosca rule:

[7] cf. atom.io

```
rule NDecl(decl⟦ rule #constructor #args? → #term* ⟧)
  → ...
```

The Tosca source code is parsed by the stable Tosca parser, while the embedded program (in `decl`) is parsed by the latest Tosca parser which potentially contains non-backward compatible modifications, e.g., changing `'→ '` to `'='`. The difference with existing approaches is Tosca parses the source program and the embedded programs at the same time. Fortunately, since Tosca compiles to Java, the typical solution consists of using multiple class loaders, allowing multiple version of the same classes—parser classes in our case— to coexist in a single JVM.

On a final note, it was very rewarding to bootstrap Tosca. By actually using it, we got a very good feeling for its strengths and conveniences—like the embedded syntax.

6 Conclusion and Future Work

We presented Tosca, a compiler generator *and* a higher-order, lazy functional programming language. In both instances the quest for a light-weight construction has been our driving force. Our research has been motivated by earlier work in the crsx project, This work emphasized the successful application of higher-order rewriting as a description language in compiler construction as well as a formal basis of a functional programming language.

The design of the Tosca compiler generator is motivated in the large by source-to-source compilers (aka transpilers). In particular it acts as a preprocessor. The source language is transformed to Java source code and an off-the-shelf Java compiler optimizes and generates executable code for us. This focus on high-level code generation allows us to obtain a sleek architecture, without having to pay the price of lower expressibility. The presented language is rooted in higher-order rewriting and features a dedicated core, which is morally equivalent to Combinatory Reduction Systems. Our programming language Tosca provides standard features of a higher-order, lazy functional language, but with a twist. Tosca features support for embedded programs, enumeration, syntactic variables, and specificity-ordered pattern matching. In particular, the latter is a unique features which allows a purely declarative programming style in the context of lazy programming, where we employ the notion of *specificity* [12].

Finally, we sketch ongoing and future work. The presentation of Tosca bypassed the incorporation of types into the programming language as well as their use in the compiler generation. In fact the produced Java code currently relies on a generic term representation, as Tosca types are discarded early during compilation. However, types naturally underlie enumerations and are in general of paramount importance in compilers. Work in progress focuses on the incorporation of a type system to Tosca so that the issued Java code actually also type-check Tosca programs. In future work we also want to incorporate automatic propagation of location information akin to the use of *string origins* in Rascal [10].

On a more theoretical level, we will work on the incorporation of ongoing research on automated resource analysis to compiler generation in general and Tosca in particular. In recent years we have seen significant advances on automated resource analysis. (We only mention recent work on the analysis of a pure, monomorphic subset of `OCaml`, cf. [5] as pointer to the literature). It is an open research question whether such static program analysis techniques can be successfully incorporated in compiler construction to guarantee efficiency of the finally emitted compiler.

References

[1] P. Aczel. A General Church-Rosser Theorem. Draft, Manchester, 1978.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracks in Theoretical Computer Science.* Cambridge University Press, 1998.

[3] A. Asperti and C. Laneve. Interaction Systems I: The Theory of Optimal Reductions. *MSCS*, 4(4):457–504, 1994.

[4] A. Asperti and C. Laneve. Interaction Systems II: The Practice of Optimal Reductions. *TCS*, 159(2):191–244, 1996.

[5] M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. 20th ICFP*, pages 152–164, 2015.

[6] Walter Bright. So You Want To Write Your Own Language?, January 2014.

[7] D. Florescu and G. Fourny. JSONiq: The History of a Query Language. *IEEE Internet Computing*, 17(5):86–90, 2013.

[8] W. D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *TCS*, 13:225–230, 1981.

[9] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[10] P. Inostroza, T. van der Storm, and S. Erdweg. Tracing Program Transformations with String Origins. In *Proc. 7th ICMT*, volume 8568 of *LNCS*, pages 154–169, 2014.

[11] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conf. on Func. Prog. Languages and Computer Architecture.*, pages 190–203. Springer-Verlag, 1985.

[12] Richard Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In Neil Jones, editor, *ESOP '90: 3rd European Symposium on Programming Copenhagen, Denmark*, pages 256–270. Springer Berlin Heidelberg, 1990.

[13] J.W. Klop. *Combinatory Reduction Systems.* PhD thesis, Utrecht University, 1980.

[14] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17:387–421, 2007.

[15] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *TCS*, 192:3–29, 1998.

[16] Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[17] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[18] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proc. OOPSLA '14*, pages 579–598. ACM, 2014.

[19] K.H. Rose. Higher-order Rewriting for Executable Compiler Specifications. In *Proc. 5th HOR*, volume 49 of *EPTCS*, pages 31–45, 2010.

[20] Kristoffer H. Rose. CRSX—Combinatory Reduction Systems with Extensions. In *Proc. 22nd RTA*, volume 10, pages 81–90. LIPIcs, 2011.

[21] Kristoffer H. Rose. Introduction to Compiler Generation Using HACS. `http://www.twosigma.com/uploads/hacs-3-3-2016.pdf`, 2016. [Online, March 2, 2016].

[22] Femke van Raamsdonk. Higher-order rewriting. In Terese, editor, *Term Rewriting Systems*, volume 55 of *Cambridge Tracks in Theoretical Computer Science*, chapter 11, pages 588–667. Cambridge University Press, 2003.

[23] P. Walmsley. *XQuery - Search Across a Variety of XML data.* O'Reilly, 2007.