A General Framework for Method Chaining Style Embedding of Domain Specific Languages

Hao Xu
University of North Carolina at Chapel Hill
xuh@cs.unc.edu

Abstract

Embedded Domain Specific Languages(DSL) are used in various programming tasks, such as SQL in database applications and regular expressions in pattern matching. Embedding DSL programs as strings in the host language that are interpreted at runtime make it extremely difficult to statically check for errors in the embedded programs. One solution is using specialized checkers, but the programming and integration of these checkers into the software development process is not easy. Another emerging paradigm is to extend the host language with support for the DSL, such as LINQ[8]. However, this requires modification of the compiler which is sometimes even more difficult, especially when the source code of the compiler is not available. Embedding a DSL into a host language using nested functions or constructors or combinators has been studied extensively[6]. However, method chaining style embedding seems to be largely ignored, probably because most of the research is done in a functional setting. As the object-oriented programming languages become more and more mature in their type systems, method chaining style may become a viable alternative as it seems to be very natural in object-oriented programming languages and widely accepted in practice[7]. In this paper, we explore the idea of type-level programming to embed DSLs into a strongly typed host language in the method chaining style. We use Scala[1, 4] to demonstrate our ideas, but these ideas should be easily applicable in other object-oriented programming languages such as Java. The ideas are partially implemented in an ongoing project – the EriLex parser generator.

Keywords Doman Specific Language, Method Chaining Style

1. Introduction

Embedded Domain Specific Languages (DSL) are used in various programming tasks, such as SQL in database applications and regular expressions in pattern matching. However, code written in these DSLs is usually embedded as a string in the host language and are interpreted at runtime. Consequently, the errors in the embedded code are not detected until runtime. One solution is checking statically for errors in the embedded code using specialized checkers. However, programming and integration of these checkers into the software development process is not easy for programmers. Another emerging paradigm is to extend the host language with support for the DSL, such as LINQ. However, this requires modification of the compiler which is sometimes even more difficult, especially when the source code of the compiler is not available. In this paper, we explore the idea of method chaining style embedding of DSLs into a strongly typed object-oriented programming language.

The idea of embedding a DSL into the host language using constructs of the host language is not new. Libraries that use this technique include various monadic parser libraries that embed BNF-like DSLs, software engineering libraries such as jMock and Hibernate Criteria Query, etc. The contribution of this paper is that we present a general framework for method chaining style embedding of DSLs into a strongly typed object-oriented programming language. We consider both the grammar and typing of the embedded language. There are several advantages to this idea in general: first, it uses the type checker of host language to check the grammar and typing of the embedded language without extra tools, which makes it less error-prone; second, given that some advanced integrated development environments such as Netbeans and Eclipse provide semantics based assistance, the embedded program can take advantage of these tools provided for the host language.

There are at least two styles of embedding a DSL. One of them embeds a DSL into a host language using nested functions or constructors or combinators, which we call the functional nesting style(FNS). FNS has been studied extensively[6]. In FNS, most terms in the object language are represented by terms in the host language, such as combinators or inductively defined data types. In contrast, the method chaining style(MCS) encodes most terms of the object language as methods in the host language, which are sometimes partial or second-class constructs in the host language, such as in Java. However there are a number of advantages in the MCS. First, sometimes the embedding of MCS is much more compact then that of FNS in an object-oriented programming language. For example to embed

```
if x then 1 else 0
```

in Java using FNS, we would write code that looks like:

```
new If(new Var("X"), new Con(1), new Con(0))
```

which is very difficult to read because of all the parentheses and keywords. Even using case classes in Scala the encoding is still a little difficult to read, because the role of `Var("X")`, `Con(1)`, and `Con(0)` are not clear from the representation.

```
If(Var("X"), Con(1), Con(0))
```

Adding constructors for `then` and `else` would further "clog" the code.

On the other hand, in MCS, we could write

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $10.00

```
If().var("X").Then().con(1).Else().con(0)
```

in Java and

```
If() varx("X") Then() con(1) Else() con(0)
```

in Scala[1], which is much easier to read.

Another advantage is that MCS makes only limited use of host language delimiters in the object language compared to FNS. for example, if we want to encode

app f **app** f X

in FNS, the encoding may look like

```
App(Var("f"), App(Var("f"), Var("X")))
```

in Scala, which uses parenthesis in the host language to delimit the pair, even when in the object language it may be clear that `app` associates to the right. In the functional programming style, usually something like

```
app (var f) ( app (var f) (var X) )
```

is necessary because unless given special rules for the `app` function in the host language,

```
app (var f) app (var f) (var X)
```

would be a totally different expression. In MCS, we can encode it in Scala as

```
app() varx("f") app() var("f") var("X")
```

which is much less relying on the host language.

The idea presented in this paper is implemented in an ongoing project – the EriLex[2] parser generator.

The paper is organized as follows: Section 2 reviews various concepts related to pushdown automata and introduces an embedding of a special class of automata into the types of Scala; Section 3 discusses practical aspects of implementing these types; Section 4 extends the embedding to deal with typing; Section 5 dicucsses related work; Section 6 discusses extensions to this framework.

2. Embedding Realtime Deterministic Pushdown Automata in Scala

Most of the definitions of grammars and automata in this section are based on [11].

Definition 2.1. A (context-free) grammar(CFG) G is a quadruple $(\mathsf{A_G}, \mathsf{Z_G}, \mathsf{z_G}, \mathsf{P_G})$, where $\mathsf{A_G}$ is a finite set of terminals, $\mathsf{Z_G}$ is a finite set of nonterminals, $\mathsf{P_G}$ a finite set of productions, and $\mathsf{z_G}$ is the start symbol. The language of a context-free grammar, written $\mathsf{L}(\mathsf{G})$, is the set of all finite length strings generated by $\mathsf{z_G}$, which is usually called a context-free language(CFL).

Let $\mathsf{a}, \mathsf{a}_1, \mathsf{a}_2 \dots$ range over $\mathsf{A_G}$, $\mathsf{z}, \mathsf{z}_1, \mathsf{z}_2 \dots$ range over $\mathsf{Z_G}$, $\mathsf{a}^n = \mathsf{a}_1^n \dots \mathsf{a}_n^n, \mathsf{i}^n = \mathsf{i}_1^n \dots \mathsf{i}_n^n, \dots$ range over $\mathsf{A_G}^n$, variables $\mathsf{z}^n = \mathsf{z}_1^n \dots \mathsf{z}_n^n, \dots$ range over $\mathsf{Z_G}^n$. Most CFG can be converted to certain equivalent normal forms. One of them is the Greibach normal form(GNF). There are usually two different definitions of GNF. One of them includes the production $\mathsf{z_G} \to \epsilon$. We use the following definition.

Definition 2.2. A CFG G is in the Greibach normal form is if all productions in $\mathsf{P_G}$ are of the form $\mathsf{z} \to \mathsf{a}\mathsf{z}^n$ where $n \in \mathbb{N}$.

Theorem 2.3. *(Greibach) Every CFL* L *where the empty string* $\epsilon \notin \mathsf{L}$ *can be generated by a CFG in the Greibach normal form.*

[1] Here we use `varx` instead of `var` because the latter is a keyword in Scala.

[2] http://erilex.kenai.com

Given a CFL G, a parse tree $\mathsf{tree}(\mathsf{a}\mathsf{s}_1 \dots \mathsf{s}_n)$ of a string $\mathsf{a}\mathsf{s}_1 \dots \mathsf{s}_n$ is a tree defined inductively

$$\mathsf{tree}(\mathsf{a}\mathsf{s}_1 \dots \mathsf{s}_n) = \mathsf{z}(\mathsf{a}, \mathsf{tree}(\mathsf{s}_1), \dots, \mathsf{tree}(\mathsf{s}_n))$$

where $\mathsf{z} \to \mathsf{a}\mathsf{z}_1 \dots \mathsf{z}_n \in \mathsf{P_G}$ and s_k is generated by z_k for all $k \in \{1, \dots, n\}$. The strings s_k are called substrings. Let $\mathsf{t}^n = \mathsf{t}_1^n \dots \mathsf{t}_n^n$ range over n subtrees and a pattern matching on trees is written $\mathsf{z}(\mathsf{a}\mathsf{t}^n)$. CFLs can be accepted by pushdown automata. Languages of CFG in GNF can be accepted by a special class of pushdown automata.

Definition 2.4. A realtime pda over an input alphabet A is (Q, Z, i, K, A, T), where Q is a finite set of states, Z a stack alphabet, T a finite subset of $A \times Q \times Z \times Z^* \times Q$ which is called the transition rules, i an initial configuration, and K a set of accepting configurations. a realtime pda is deterministic, or a dpda, if and only if T is a function $A \times Q \times Z \to Z^* \times Q$.

The different between a realtime pda and a pda is that realtime pdas do not have ϵ-rules. Because any proper context-free language that does not contains ϵ can be generated by a CFG in GNF without ϵ-productions, the languages accepted by realtime pdas differ from those accepted by pdas only in whether ϵ may be included, which is not a significant restriction in DSL design.

What is interesting is the difference between realtime deterministic pda and deterministic pda. Although dpda is strictly less powerful than pda, it has been shown that languages generated by dpda are exactly LR(k), which contains most programming languages. Therefore, dpda is powerful enough for our purpose. But realtime dpda is strictly less powerful than dpda. There are languages that are accepted by some dpda but no realtime dpda. For example, the language $L = \{<^p \ \{^n >^p \ | p, n > 0\} \cup \{<^p \ \{^n \}^n | p, n > 0\}$, which is a modification of an example in [11], can be accepted by a dpda but no realtime dpda. The problem is that in order to make sure that at lease one of ">" or "}" matches with "<" or "{", we need to push state symbols on the stack to match the number of "<"'s and "{"'s. If ">" is encountered, we need to remove those symbols that are generated by "{". However, this is a very rare situation in language design, because here we want to design a language that does not keep all brackets matched on purpose. If the object language has a universal hierarchy of programming constructs in which programming constructs in the lower level are strictly and properly contained (i.e. no overlapping boundary) by programming constructs in the higher level, as in Java where the delimiter such as "{" and "}" are used verbosely, then realtime dpda is expressive enough because the number of pops is always bounded. We work with dpda in the following sections.

Next, we look at how to embed a realtime dpda in Scala. Before proceeding to define the embedding, we first present an equivalent form of definition of a realtime dpda.

Definition 2.5. A stateless realtime dpda(slrpda) $\mathcal{A}$ over an input alphabet A is $(\mathsf{Z}, \mathsf{z}_\mathcal{A}, \mathsf{A}, \mathcal{T})$, where Z is a stack alphabet, $\mathcal{T}$ a function $\mathsf{A} \times \mathsf{Z} \to \mathsf{Z}^*$ which is called the transition rules, and $\mathsf{z}_\mathcal{A}$ an initial configuration, with ϵ as an accepting configuration. The language accepted by $\mathcal{A}$ is the set of all strings over A accepted by it.

It is easy to see that we can construct an equivalent slrpda from every realtime dpda by encoding the current state in the top stack symbol.

Let $\mathcal{A}$ be a slrpda. We assume that a name embedding function $\mathsf{e} : A \cup Z \to \mathsf{ScalaName}$ that maps injectively slrpda symbols to valid Scala names is predefined. We write $\mathsf{e}(\mathsf{a})$ as $\mathsf{e_a}$, and $\mathsf{e}(\mathsf{z})$ as $\mathsf{e_z}$.

To simplify the presentation, we may write a Scala class

$$\textbf{class}\ c\{def_1\ def_2\ \dots\ def_n\}$$

as

$$\textbf{class}\ c\{def_1\}, \textbf{class}\ c\{def_2\}, \ldots, \textbf{class}\ c\{def_n\}$$

where c is a class signature and $def_1, def_2, \ldots, def_n$ are method definitions. When combining $\textbf{class}\ c_1\{def_1\}, \ldots, \textbf{class}\ c_n\{def_n\}$ where $c_1, \ldots, c_n$ all have the same type constructor, back to $\textbf{class}\ c\{def_1\ def_2\ \ldots\ def_n\}$, we may need to rename the type parameters in $c_1, \ldots, c_n$, which can be done using simultaneous unification. For example, $\textbf{class}\ A[\tau_1]\{def_1\}, \ldots, \textbf{class}\ A[\tau_n]\{def_n\}$ are combined to $\textbf{class}\ A[\tau]\{def_1, \ldots, def_n\}$. For succinctness, we write type $Null$ as $\bot$.

Definition 2.6. The embedding of an slrpda $\mathcal{A} = (\mathsf{Z}, \mathsf{z}_{\mathcal{A}}, \mathsf{A}, \mathcal{T})$, written $\mathsf{en}^{\mathsf{A}}_{\mathsf{L}}(\mathcal{A}, \mathsf{e})$, is a set of classes with the name of $\mathsf{e_z}$ for all $\mathsf{z} \in Z$.

$$\begin{aligned} \mathsf{en}^{\mathsf{A}}_{\mathsf{L}}(\mathcal{A}, \mathsf{e}) \quad = \quad & \{\textbf{abstract class}\ \mathsf{e_z}[\kappa]\{ \\ & \quad \textbf{def}\ \mathsf{e_a}() : \mathsf{e}_{\mathsf{z}_1}[\ldots \mathsf{e}_{\mathsf{z}_n}[\kappa]]\} \\ & \quad |(\mathsf{a}, \mathsf{z}, \mathsf{z}_1 \ldots \mathsf{z}_n) \in \mathcal{T}\} \end{aligned}$$

The embedding of an input string a^n, written $\mathsf{en}^{\mathsf{A}}_{\mathsf{S}}(\mathsf{a}^n)$, is the method chain

$$\mathsf{en}^{\mathsf{A}}_{\mathsf{S}}(\mathsf{a}^n, \mathsf{e}) \quad = \quad \mathsf{e}_{\mathsf{a}^n_1}() \ldots \mathsf{e}_{\mathsf{a}^n_n}()$$

The well-formness is summarized in the following theorem.

Theorem 2.7. *If $\mathcal{A}$ is a slrpda and $\mathsf{imp}_{\mathsf{z}_{\mathcal{T}}}$ is an implementation of $\mathsf{e}_{\mathsf{z}_{\mathcal{T}}}$, then $\mathsf{a}^n \in \mathsf{L}(\mathcal{A})$ if and only if $\textbf{new}\ \mathsf{imp}_{\mathsf{z}_{\mathcal{T}}}[\bot]()\mathsf{en_S}(\mathsf{a}^n, \mathsf{e})$ has type $\bot$ in Scala.*

Proof. (Sketch) [4] did not deal with generics, but we can still use the (METHOD) rule, by viewing the type metavariables as also ranging over constructed types. Given a sequence of input a^n, we construct a sequence of configurations of the slrpda. Also, given the method chain, we construct a sequence of types by instantiating the (METHOD) rule. The initial empty stack is represented by $\bot$. By induction, we can prove that the each configuration is represented by the corresponding type. Because the type of the Scala expression is $\bot$, the input is accepted by the slrpda. ☐

A grammar in the Greibach normal form(GNF) and LL(1) (i.e. no common prefix) can be converted directly to an slrpda. The slrpda is quite straightforward to construct, by just using nonterminals as stack symbols. The embedding $\mathsf{en}^{\mathsf{G}}_{\mathsf{L}}(\mathsf{G})$ of a LL(1) GNF grammar G is defined as

Definition 2.8. The embedding of G in LL(1) GNF w.r.t name embedding function e, written $\mathsf{en}^{\mathsf{G}}_{\mathsf{L}}(\mathsf{G}, \mathsf{e})$, is a set of classes with the name of $\mathsf{e_z}$ for all $\mathsf{z} \in Z$.

$$\begin{aligned} \mathsf{en}^{\mathsf{G}}_{\mathsf{L}}(\mathsf{G}, \mathsf{e}) \quad = \quad & \{\textbf{abstract class}\ \mathsf{e_z}[\kappa]\{ \\ & \quad \textbf{def}\ \mathsf{e_a}() : \mathsf{e}_{\mathsf{z}^n_1}[\ldots \mathsf{e}_{\mathsf{z}^n_n}[\kappa]]\} \\ & \quad |\mathsf{z} \rightarrow \mathsf{a}\mathsf{z}^n \in P\} \end{aligned}$$

The embedding of an input string a^n, written $\mathsf{en}^{\mathsf{G}}_{\mathsf{S}}(\mathsf{a}^n, \mathsf{e})$, is the method chain

$$\mathsf{en}^{\mathsf{G}}_{\mathsf{S}}(\mathsf{a}^n, \mathsf{e}) \quad = \quad \mathsf{e}_{\mathsf{a}^n_1}() \ldots \mathsf{e}_{\mathsf{a}^n_n}()$$

Theorem 2.9. *If G is a grammar in LL(1) GNF and $\mathsf{imp}_{\mathsf{z}_{\mathsf{G}}}$ is an implementation of $\mathsf{e}_{\mathsf{z}_{\mathsf{G}}}$, then $\mathsf{a}^n \in \mathsf{L}(\mathsf{G})$ if and only if $\textbf{new}\ \mathsf{imp}_{\mathsf{z}_{\mathsf{G}}}[\bot]()\mathsf{en}^{\mathsf{G}}_{\mathsf{S}}(\mathsf{a}^n)$ has type $\bot$ in Scala.*

Proof. (Sketch) Prove by converting the grammar to an equivalent slrpda. ☐

Next, let's look at an example.

$$\begin{aligned} & \textbf{abstract class}\ \mathsf{e}_s[\kappa]\{ \\ & \quad \textbf{def}\ \mathsf{e}_(() : \mathsf{e}_r[\kappa] \\ & \} \\ & \textbf{abstract class}\ \mathsf{e}_r[\kappa]\{ \\ & \quad \textbf{def}\ \mathsf{e}_a() : \mathsf{e}_{r'}[\kappa] \\ & \quad \textbf{def}\ \mathsf{e}_(() : \mathsf{e}_r[\mathsf{e}_{r'}[\kappa]] \\ & \} \\ & \textbf{abstract class}\ \mathsf{e}_{r'}[\kappa]\{ \\ & \quad \textbf{def}\ \mathsf{e}_)() : \kappa \\ & \quad \textbf{def}\ \mathsf{e}_*() : \mathsf{e}_{r'}[\kappa] \\ & \quad \textbf{def}\ \mathsf{e}_a() : \mathsf{e}_{r'}[\kappa] \\ & \quad \textbf{def}\ \mathsf{e}_(() : \mathsf{e}_r[\mathsf{e}_{r'}[\kappa]] \\ & \} \end{aligned}$$

Figure 1. Embedding Scala Classes

Example 2.10. Let Σ be a finite alphabet. We can embed in Scala the regular expression DSL given by $s \rightarrow (r), r \rightarrow a|r*|rr|r'|'r|(r)$, where $a \in \Sigma$. We added an additional symbol s, which makes the productions much easier to work with. First, we convert it to an grammar in LL(1) GNF.

$$\begin{array}{lllllll} r & \rightarrow & ar' & \quad & r' & \rightarrow & *r' \\ r & \rightarrow & (rr' & & r' & \rightarrow & '|'r \\ r' & \rightarrow &) & & r' & \rightarrow & (rr' \\ r' & \rightarrow & ar' & & s & \rightarrow & (r \end{array}$$

We can generate the embedding directly from the GNF. For example, for the first production, we have $\textbf{class}\ \mathsf{e}_r[\kappa]\{\textbf{def}\ a() : \mathsf{e}_{r'}[\kappa]\}$. When combined together, the embedding classes are in the Figure 1.

If imp_s is an implementation of e_s, then for a regular expression $(a*)$,

$$\textbf{new}\ \mathsf{imp}_s[\bot]()\mathsf{e}_(()\mathsf{e}_a()\mathsf{e}_*()\mathsf{e}_)()$$

has type $\bot$; and for a string $|a|$,

$$\textbf{new}\ \mathsf{imp}_s[\bot]()\mathsf{e}_|()\mathsf{e}_a()\mathsf{e}_|()$$

is rejected by Scala.

The intuition behind this embedding is that a type $\mathsf{e_z}[\mathsf{e}_{\mathsf{z}_1}[\ldots \mathsf{e}_{\mathsf{z}_n}[\bot]]]$ represents the configuration of the parser as a stack of the type (constructors). The current parser state is parsing nonterminal z. When z is successfully parsed, the parser continues to parse z_1, and so forth. When parsing for some z in the grammar, the leading terminals decide which productions to enter and which nonterminals to push onto the stack that represents the next configuration of the parser. The choice is always deterministic because the grammar is LL(1).

Definition 2.11. We call the stack of type (constructors) representing the current configuration of the slrpda the continuation stack. The top of continuations stack refers to the outermost type (constructor).

Suppose that the continuation stack is $\mathsf{e_z}[\bot]$. If the selected production is of the form $\mathsf{z} \rightarrow \mathsf{a}$, then it just removes the current symbol from the top of the continuation stack by setting the return type of the method the type κ, and the new stack is represented by type κ as shown in Figure 2(a). If the selected production of the form $\mathsf{z} \rightarrow \mathsf{a}\mathsf{z}_1\mathsf{z}_2$, then it pushes the type constructors representing $\mathsf{z}_1, \mathsf{z}_2$ onto the continuation stack as shown in Figure 2(b).

3. Untyped MCS

In the previous section, we defined an embedding function en^{G} that embeds a grammar in LL(1) GNF into abstract classes of Scala. But in this embedding the embedding classes can not be instantiated because they are abstract classes. In this section, we discuss

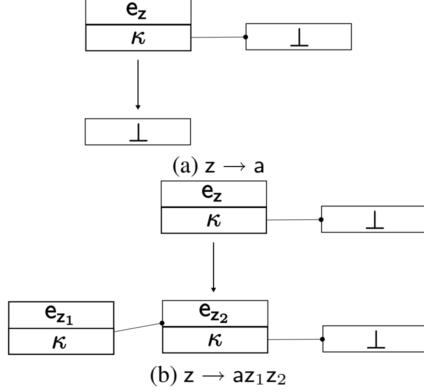


Figure 2. Examples

systematically the methodology of embedding that generates concrete classes.

3.1 Methodology

The methodology of embedding is that each embedding en of a grammar G is a pair of embedding functions (en_L, en_S) parameterized over G and a name embedding function e, where en_L is a *language embedding* that generates support classes from G, and en_S is a *string embedding* that generates an expression for a string $s \in L(G)$. And the soundness theorem will be in the form that if G satisfies certain properties, then with embedding $en_L(G)$, a string $s \in L(G)$ if and only if the embedding $en_S(s)$ of s is well-typed in Scala.

3.2 Workaround for Restrictions in Scala

Before we look at the embedding, let us look at a restriction of generics in Scala and how to work around it. One of the restrictions in languages that use erasure such as Scala and Java is that type parameters can not be used to create an object because they are erased during compilation. For example, **new** $\tau()$ where τ is a type parameter is not allowed. This is in contrast to C++ templates and C# generics, which expand the generic types to instances of these types during compilation or at runtime.

Example 3.1. We implement the abstract classes in the previous example. In a first attempt, we may be tempted to implement the class $e_{r'}[\kappa]$ as

$$\begin{array}{l}\textbf{class } e_{r'}[\kappa]\{\\ \quad \textbf{def } e_{)}() = \textbf{new } \kappa()\\ \quad \ldots\\ \}\end{array}$$

but this is not a valid Scala program! This problem occurs for methods that represents productions of the form $z \longrightarrow a$. We can work around this restriction by letting the creator of this class pass in a continuation that knows how to create an object of type κ. In this example, we can simply pass in an object of type κ.

$$\begin{array}{l}\textbf{class } e_{r'}[\kappa](k:\kappa)\{\\ \quad \textbf{def } e_{)}() = k\\ \quad \ldots\\ \}\end{array}$$

The implementation of other methods are straightforward. The implementation of $e_s, e_r, e_{r'}$ is shown in Figure 3. In methods that add more than one type constructors the type of the continuation, such as $e_{(}()$, we need to rewire the continuation properly. Now, in order to initialize the parser, we set the type of the initial continuation to $\bot$ and pass in an intial continuation $null$ which is the only object of that type .

$$\begin{array}{l}\textbf{class } e_s[\kappa](k:\kappa)\{\\ \quad \textbf{def } e_{(}() = \textbf{new } e_r[\tau](k)\\ \}\\ \textbf{class } e_r[\kappa](k:\kappa)\{\\ \quad \textbf{def } e_a() = \textbf{new } e_{r'}[\kappa](k)\\ \quad \textbf{def } e_{(}() = \textbf{new } e_r[e_{r'}[\kappa]](\textbf{new } e_{r'}[\kappa](k))\\ \}\\ \textbf{class } e_{r'}[\kappa](k:\kappa)\{\\ \quad \textbf{def } e_a() = \textbf{new } e_{r'}[\kappa](k)\\ \quad \textbf{def } e_{*}() = \textbf{new } e_{r'}[\kappa](k)\\ \quad \textbf{def } e_{(}() = \textbf{new } e_r[e_{r'}[\kappa]](\textbf{new } e_{r'}[\kappa](k))\\ \quad \textbf{def } e_{)}() = k\\ \}\end{array}$$

Figure 3. Implementation

$$\begin{array}{rcl}\kappa_0^n & \triangleq & \kappa\\ \kappa_{i+1}^n & \triangleq & e_{z_{n-i}^n}[\kappa_i^n]\\ k_0^n & \triangleq & k\\ k_{i+1}^n & \triangleq & \textbf{new } e_{z_{n-i}^n}(k_i^n)\end{array}$$

(a)

$$\begin{array}{rcl}en_L^l(G,e) & = & \{en_L^l(p)|p \in P_G\}\\ en_L^l(z \longrightarrow az^n, e) & = & \textbf{class } e_z[\kappa](k:\kappa)\{en_L^l(az^n)\}\\ en_L^l(az^n, e) & = & \textbf{def } e_a() : \kappa_n^n = k_n^n\end{array}$$

(b)

$$en_S^l(G, a^n, e) = \textbf{new } e_{z_G}[\bot](null)e_{a_1^n}()\ldots e_{a_n^n}() : \bot$$

(c)

Figure 4. Embedding Rules

3.3 The General Embedding Functions

The embedding en^l embeds a grammar G as concrete classes. The definition of the language embedding en_L^l is given in Figure 4(a) and Figure 4(b). The rules represents a production of the form $z \longrightarrow a$ by a method that removes the type constructor at the top of the continuation stack by returning the continuation k of type κ and represents a production of the form $z \longrightarrow az_1 \ldots z_n$ by a method that pushes the type constructors representing $z_1, z_2, \ldots, z_n$ onto the continuation stack by constructing objects **new** $e_{z_1}(\ldots e_{z_n}(k)))$. If written separately, they are

$$\begin{array}{rcl}{en_L^l}'(a) & = & \textbf{def } e_a() : \tau = \kappa\\ {en_L^l}'(az_1\ldots z_n) & = & \textbf{def } e_a() : e_{z_1}[\ldots e_{z_n}[\kappa]]] =\\ & & \quad \textbf{new } e_{z_1}(\ldots e_{z_n}(k)))\end{array}$$

The string embedding en_S^l is given in Figure 4(c).

Corollary 3.2. *If G is a grammar in LL(1) GNF, , e is a name embedding function, then $a^n \in L(G)$ if and only if $en_L^l(G,e) \cup \{en_S^l(G,a^n,e)\}$ is well-typed in Scala.*

3.4 Parametrized Grammar

In this section, we introduce an extension to grammars, named parametrized grammars, that will be used in this paper.

Definition 3.3. A parametrized grammar is a grammar $G = (Z_G, z_G, A_G, P_G)$ in the GNF with a mapping $\mathsf{arity} : P_G \rightarrow \mathbb{N}$.

For each production $\mathsf{z} \rightarrow \mathsf{a}\mathsf{z}^n$ of the grammar, arity assigns an integer k such that $0 \leq k \leq n$. The first k nonterminals on the right hand side of the production are called parameters of a. Furthermore, we require that $\mathsf{Z_G}$ can be divided into two disjoint subsets $\mathsf{Z}_{\mathsf{G}_0}, \mathsf{Z}_{\mathsf{G}_1}$ such that in all productions for $\mathsf{z} \in \mathsf{Z}_{\mathsf{G}_1}$, any $\mathsf{z}_0 \in \mathsf{Z}_{\mathsf{G}_0}$ only appears as parameters and for all productions for $\mathsf{z}_0 \in \mathsf{Z}_{\mathsf{G}_0}$, no $\mathsf{z} \in \mathsf{Z}_{\mathsf{G}_1}$ or parameter appears.

Effectively, we divided $\mathsf{Z_G}$ into two levels. The terminals in productions of top level nonterminals may take parameters from the second level nonterminals. And the second level can be written as a set of inductively defined data types, with the terminals as constructors and nonterminals as types. We use $\rightarrow$ to mark top level symbols and $=$ to mark second level symbols. Given a parametrized grammar G, we write the productions for $\mathsf{Z}_{\mathsf{G}_0}$ as $\mathsf{P}_{\mathsf{G}_0}$ and productions for $\mathsf{Z}_{\mathsf{G}_1}$ as $\mathsf{P}_{\mathsf{G}_1}$.

Example 3.4. The untyped lambda calculus with de Brujin index:

$$\begin{array}{lcl} e & \rightarrow & var(i) | app\ e\ e | abs\ e \\ i & = & z | s\ i \end{array}$$

e is a top level nonterminal, while i is a second level nonterminal. The terminal var takes i as a parameter.

We will encode second level nonterminals as method parameters in the embedding, which is summarized in the following definition.

To disambiguate with name embedding function e which are used to encode top level symbols, we use another name embedding functions d to encode second level symbols. The embedding en^0_L of second level symbols of a parametrized grammar G with regard to the name embedding function d is a set of case classes (or inductive data types, as sometimes they are called) defined in Figure 5(a). The embedding en^1_L of top level symbols of a parametrized grammar G with regard to the name embedding functions e and d is a set of classes with the name $\mathsf{e_z}$ for all $\mathsf{z} \in \mathsf{Z}_{\mathsf{G}_1}$ as defined in Figure 5(b). The string embedding of a string $\mathsf{a}_1 \ldots \mathsf{a}_n$ generated by some $\mathsf{z} \in \mathsf{Z}_{\mathsf{G}_0}$ is inductively defined on its parse tree where t^n_k is a subtree for $k \in \{1, \ldots, n\}$ as shown in Figure 5(c). The string embedding of string $\mathsf{a}_1\mathsf{i}^{k_1} \ldots \mathsf{a}_n\mathsf{i}^{k_n}$, where i^{k_i} for $i \in \{1, \ldots, n\}$ are strings generated by some $\mathsf{z} \in Z_0$, is a method chain as shown in Figure 5(d).

3.5 Building Abstract Syntax Trees

In this section we discuss how to modify the parser to build an abstract syntax tree when the parser is run. We first look at another example,

Example 3.5. Since the grammar in Example 3.4 is already in GNF, we can embed it as classes in Scala using en^l. We use the following name embedding: e map var to $varx$, app to app, abs to abs, and e to E; d maps e to $Term$, var to Var, app to App, abs to Abs, i to Ref, z to $Zero$, and s to $Succ$.

$$\begin{array}{l} \textbf{class}\ E[\kappa](k:\kappa)\{ \\ \quad \textbf{def}\ varx(i:Ref):\kappa = k \\ \quad \textbf{def}\ app():E[E[\kappa]] = \textbf{new}\ E(E(k)) \\ \quad \textbf{def}\ abs():E[\kappa] = \textbf{new}\ E(k) \\ \} \end{array}$$

We extend embedding en^0 in Section 3.4 to top level symbols to generate data structures for storing the abstract syntax tree. A string generated by e is represented by $Term$, and a string generated by i is represented by Ref, as shown in Figure 6(a). Next, we add a builder function to the implementation as shown in Figure 6(b).

We made three modifications here. First, a new type parameter σ is added to the class indicating the type of the value built by the builder. Second, a parameter is added to the class $b : Term \Rightarrow \sigma$, which is the builder. Third, the type of k is modified to $\sigma \Rightarrow \tau$. The reason is that we need to pass the builder to the continuation, so that the currently building tree is part of the continuation constructed by k during runtime.

$$\begin{array}{rcl} \mathsf{en}^0_\mathsf{L}(\mathsf{G},\mathsf{d}) & = & \{\mathsf{en}^0_\mathsf{L}(\mathsf{p},\mathsf{d}) | \mathsf{p} \in \mathsf{P}_{\mathsf{G}_0}\} \cup \{\mathsf{en}^0_\mathsf{L}(\mathsf{z},\mathsf{d}) | \mathsf{z} \in \mathsf{Z}_{\mathsf{G}_0}\} \\ \mathsf{en}^0_\mathsf{L}(\mathsf{z} \rightarrow \mathsf{a}\mathsf{z}^n,\mathsf{d}) & = & \textbf{case class}\ \mathsf{d_a}(i_1:\mathsf{d}_{\mathsf{z}^n_1},\ldots,i_n:\mathsf{d}_{\mathsf{z}^n_n}) \\ & & \quad \textbf{extends}\ \mathsf{d_z}() \\ \mathsf{en}^0_\mathsf{L}(\mathsf{z},\mathsf{d}) & = & \textbf{abstract case class}\ \mathsf{d_z}() \end{array}$$

(a)

$$\begin{array}{rcl} \mathsf{en}^1_\mathsf{L}(\mathsf{G},\mathsf{d},\mathsf{e}) & = & \{\mathsf{en}^1_\mathsf{L}(\mathsf{p},\mathsf{d},\mathsf{e}) | \mathsf{p} \in \mathsf{P}_{\mathsf{G}_1}\} \\ \mathsf{en}^1_\mathsf{L}(\mathsf{z} \rightarrow \mathsf{a}\mathsf{z}^p\mathsf{z}^n,\mathsf{d},\mathsf{e}) & = & \textbf{class}\ \mathsf{e_z}[\kappa]\{ \\ & & \quad \textbf{def}\ \mathsf{e_a}(i_1:\mathsf{d}_{\mathsf{z}^p_1},\ldots,i_p:\mathsf{d}_{\mathsf{z}^p_p}): \\ & & \quad\quad \kappa^n_n = \mathsf{k}^n_n\} \end{array}$$

(b)

$$\begin{array}{rcl} \mathsf{en}^0_\mathsf{S}(\mathsf{a}^n,\mathsf{G},\mathsf{d}) & = & \mathsf{en}^{0\prime}_\mathsf{S}(\mathsf{tree}(\mathsf{a}^n),\mathsf{G},\mathsf{d}) \\ \mathsf{en}^{0\prime}_\mathsf{S}(\mathsf{z}(\mathsf{a}\mathsf{t}^n),\mathsf{G},\mathsf{d}) & = & \mathsf{d_a}(\mathsf{en}^{0\prime}_\mathsf{S}(\mathsf{t}^n_1,\mathsf{G},\mathsf{d}),\ldots\mathsf{en}^{0\prime}_\mathsf{S}(\mathsf{t}^n_n,\mathsf{G},\mathsf{d})) \end{array}$$

(c)

$$\begin{array}{rcl} \mathsf{en}^1_\mathsf{S}(\mathsf{a}_1\mathsf{i}^{k_1}\ldots\mathsf{a}_n\mathsf{i}^{k_n},\mathsf{G},\mathsf{d},\mathsf{e}) & = & \textbf{new}\ \mathsf{e}_{\mathsf{z_G}}[\bot]()\mathsf{s}_1\ldots\mathsf{s}_n:\bot \\ \text{where} & & \\ \mathsf{s}_i & = & \mathsf{e}_{\mathsf{a}_i}(\mathsf{en}^0_\mathsf{S}(\mathsf{i}^{k_i}_1,\mathsf{G},\mathsf{d}), \\ & & \quad \ldots,\mathsf{en}^0_\mathsf{S}(\mathsf{i}^{k_i}_{k_i},\mathsf{G},\mathsf{d})) \end{array}$$

(d)

Figure 5. Embedding Rules

$$\begin{array}{l} \textbf{abstract case class}\ Term() \\ \textbf{case class}\ Var(x:Ref)\ \textbf{extends}\ Term() \\ \textbf{case class}\ App(t_1:Term,t_2:Term)\ \textbf{extends}\ Term() \\ \textbf{case class}\ Abs(t:Term)\ \textbf{extends}\ Term() \\ \\ \textbf{abstract case class}\ Ref() \\ \textbf{case class}\ Zero()\ \textbf{extends}\ Ref() \\ \textbf{case class}\ Succ(r:Ref)\ \textbf{extends}\ Ref() \end{array}$$

(a)

$$\begin{array}{l} \textbf{class}\ E[\kappa,\sigma](b:Term \Rightarrow \sigma, k:\sigma \Rightarrow \kappa)\{ \\ \ \textbf{def}\ varx(i:Ref):\kappa = k(b(Var(i))) \\ \ \textbf{def}\ app():E[E[\kappa,\sigma],Term \Rightarrow \sigma] = \textbf{new}\ E(\\ \quad\quad (t_1:Term) \Rightarrow (t_2:Term) \Rightarrow b(App(t_1,t_2)), \\ \quad\quad (b_1:Term \Rightarrow \sigma) \Rightarrow E(b_1,k)) \\ \ \textbf{def}\ abs():E[\kappa,\sigma] = \textbf{new}\ E((t:Term) \Rightarrow b(Abs(t)),k) \\ \} \end{array}$$

(b)

Figure 6. Implementation with a Builder Function

Unlike in FNS, sometimes we need to construct a partial AST whose subtrees are filled in later. Partial trees are built by constructing new builders. For example, the app function constructs the builder $(t_1 : Term) \Rightarrow (t_2 : Term) \Rightarrow b(App(t_1, t_2))$ and passes it to the continuation k, which encodes how to build the App object. But the actual construction of the App node happens when all parameters to the constructor are built.

To use this class, we need to pass in an initial builder to the constructor of the class where we create an object of the class.

Using the new embedding, we can encode terms of $\mathsf{L}(\mathsf{G})$ in Scala. We define an auxiliary Scala method.

$$\mathbf{def}\ term() = \mathbf{new}\ E((t : Term) \Rightarrow null, (s : Term) \Rightarrow s)$$

Using this method, the following term

$$app\ abs\ varx(z)\ con(1)$$

is encoded in Scala as

$$term()\ app()\ abs()\ varx(Zero())\ con(1) : \bot$$

and the following term

$$app\ app\ abs\ abs\ con(2)\ app\ abs\ varx(z)\ con(1)\ con(1)$$

is encoded in Scala as

```
term()
  app()
    app()
      abs()
        abs()
          con(2)
      app()
        abs()
          varx(Zero())
        con(1) : ⊥
```

Next, we formalize the ideas of Example 3.5 in the embedding en^B. en_B is defined as a pair of embeddings $(\mathsf{en}^\mathsf{d}, \mathsf{en}^\mathsf{b})$ which are parametrized over grammar G, name embedding function e which maps all symbols in G to a unique valid Scala names, and name embedding function d which maps all symbols in G to unique valid Scala names such that $\mathsf{d}_z \neq \mathsf{e}_z$ for all $\mathsf{z} \in \mathsf{A}_\mathsf{G} \cup \mathsf{Z}_\mathsf{G}$. e is used in en^b while d is used both in en^b and en^d. en^d is an extension of en^0 to all top level productions p and for all top level nonterminals z, as shown in Figure 7(a) and Figure 7(b). en^b is an extension of en^1 to include the builder function. We use the auxiliary definitions as shown in Figure 7(c). σ_n^n is the type for the n-ary builders that build values of type σ. p is the number of parameters of a. $\mathsf{B}_n^{n,p}$ is an n-ary builder and its type is always σ_n^n. The degenerated builder $\mathsf{B}_0^{0,p}$ is actually the value built by the builder. κ_n^n is the continuation stack. k_n^n is the continuation whose type is always $\sigma_n^n \Rightarrow \kappa_n^n$. z^p are parameters. The rules are shown in Figure 7(d). The string embedding $\mathsf{en}_\mathsf{S}^\mathsf{b}$ of the object language is defined in Figure 7(e), where a_k is a top level symbols for $k \in \{1, \ldots, n\}$ and i^{k_i} are second level strings for $i \in \{1, \ldots, n\}$. The embedding en^B is defined in Figure 7(f).

We need the following theorem to complete our discussion.

Theorem 3.6. *If G is a grammar of the terms of the object language in LL(1) GNF, e maps $\mathsf{A}_\mathsf{G} \cup \mathsf{Z}_\mathsf{G}$ in G to unique Scala names, d maps $\mathsf{A}_\mathsf{G} \cup \mathsf{Z}_\mathsf{G}$ to unique Scala name such that $\mathsf{d}_\mathsf{z} \neq \mathsf{e}_\mathsf{z}$ for all $\mathsf{z} \in \mathsf{Z}_\mathsf{G}$. If a^n is a term of the object language, then $\mathsf{a}^n \in \mathsf{L}(\mathsf{G})$ if and only if $\mathsf{en}_\mathsf{L}^\mathsf{B}(\mathsf{G},\mathsf{d},\mathsf{e}) \cup \{\mathsf{en}_\mathsf{S}^\mathsf{B}(\mathsf{a}^n,\mathsf{G},\mathsf{d},\mathsf{e})\}$ is well-typed in Scala.*

Proof. (Sketch) Note that any encoded term is automatically well-formed because the constructor d_a only encodes well-formed terms. First, we prove that all encoded terms are well-formed. Then, by the type safety, the encoding always constructs encoding of some well-formed term. Finally, we prove that the term constructed is the same term that is encoded in the method chain. □

$$\begin{aligned}
\mathsf{en}_\mathsf{L}^\mathsf{d}(\mathsf{G},\mathsf{d}) &= \{\mathsf{en}_\mathsf{L}^\mathsf{d}(\mathsf{p},\mathsf{d}) | \mathsf{p} \in \mathsf{P}_\mathsf{G}\} \cup \{\mathsf{en}_\mathsf{L}^\mathsf{d}(\mathsf{z},\mathsf{d}) | \mathsf{z} \in \mathsf{Z}_\mathsf{G}\} \\
\mathsf{en}_\mathsf{L}^\mathsf{d}(\mathsf{z} \to \mathsf{a}\mathsf{z}^n, \mathsf{d}) &= \mathbf{case\ class}\ \mathsf{d}_\mathsf{a}(i_1 : \mathsf{d}_{\mathsf{z}_1^n}, \ldots, i_n : \mathsf{d}_{\mathsf{z}_n^n}) \\
&\qquad \mathbf{extends}\ \mathsf{d}_\mathsf{z}() \\
\mathsf{en}_\mathsf{L}^\mathsf{d}(\mathsf{z},\mathsf{d}) &= \mathbf{abstract\ case\ class}\ \mathsf{d}_\mathsf{z}()
\end{aligned}$$

(a)

$$\begin{aligned}
\mathsf{en}_\mathsf{S}^\mathsf{d}(\mathsf{a}^n,\mathsf{d}) &= \mathsf{en}_\mathsf{S}^{\mathsf{d}\prime}(\mathsf{tree}(\mathsf{a}^n),\mathsf{d}) \\
\mathsf{en}_\mathsf{S}^{\mathsf{d}\prime}(\mathsf{z}(\mathsf{a}\mathsf{t}^n),\mathsf{d}) &= \mathsf{d}_\mathsf{a}(\mathsf{en}_\mathsf{S}^{\mathsf{d}\prime}(\mathsf{t}_1^n,\mathsf{d}), \ldots \mathsf{en}_\mathsf{S}^{\mathsf{d}\prime}(\mathsf{t}_n^n,\mathsf{d}))
\end{aligned}$$

(b)

$$\begin{aligned}
\sigma_0^n &\triangleq \sigma \\
\sigma_{i+1}^n &\triangleq \mathsf{d}_{\mathsf{z}_{n-i}^n} \Rightarrow \sigma_i^n \\
\kappa_0^n &\triangleq \kappa \\
\kappa_{i+1}^n &\triangleq \mathsf{e}_{\mathsf{z}_{n-i}^n}[\kappa_i^n, \sigma_i^n] \\
\mathsf{k}_0^n &\triangleq k \\
\mathsf{k}_{i+1}^n &\triangleq (b_{i+1} : \sigma_{i+1}^n) \Rightarrow \mathbf{new}\ \mathsf{e}_{z_{n-i}^n}(b_{i+1}, \mathsf{k}_i^n) \\
\mathsf{B}_0^{n,p} &\triangleq b(\mathsf{e}_\mathsf{a}(i_1, \ldots, i_p, t_1, \ldots, t_n)) \\
\mathsf{B}_{i+1}^{n,p} &\triangleq (t_{n-i} : \mathsf{d}_{\mathsf{z}_{n-i}^n}) \Rightarrow \mathsf{B}_i^{n,p}
\end{aligned}$$

(c)

$$\begin{aligned}
\mathsf{en}_\mathsf{L}^\mathsf{b}(\mathsf{G},\mathsf{d},\mathsf{e}) &= \{\mathsf{en}_\mathsf{L}^\mathsf{b}(\mathsf{p},\mathsf{d},\mathsf{e}) | \mathsf{p} \in \mathsf{P}_{\mathsf{G}_1}\} \\
\mathsf{en}_\mathsf{L}^\mathsf{b}(\mathsf{z} \to \mathsf{a}\mathsf{z}^p\mathsf{z}^n, \mathsf{d}, \mathsf{e}) &= \mathbf{class}\ \mathsf{e}_\mathsf{z}[\kappa]\{ \\
&\qquad \mathbf{def}\ \mathsf{e}_\mathsf{a}(i_1 : \mathsf{d}_{\mathsf{z}_1^p}, \ldots, i_p : \mathsf{d}_{\mathsf{z}_p^p}) : \\
&\qquad\quad \kappa_n^n = \mathsf{k}_n(\mathsf{B}_n^{n,p})\}
\end{aligned}$$

(d)

$$\begin{aligned}
\mathsf{en}_\mathsf{S}^\mathsf{b}(\mathsf{a}_1\mathsf{i}^{k_1} \ldots \mathsf{a}_n\mathsf{i}^{k_n}, \mathsf{G},\mathsf{d},\mathsf{e}) &= \mathbf{new}\ term()\mathsf{s}_1 \ldots \mathsf{s}_n : \bot \\
\text{where}& \\
\mathsf{s}_i &= \mathsf{e}_{\mathsf{a}_i}(\mathsf{en}_\mathsf{S}^\mathsf{d}(\mathsf{i}_1^{k_i},\mathsf{d}), \\
&\qquad \ldots, \mathsf{en}_\mathsf{S}^\mathsf{d}(\mathsf{i}_{k_i}^{k_i},\mathsf{d})) \\
\mathbf{def}\ term() &= \mathbf{new}\ \mathsf{e}_{\mathsf{z}_\mathsf{G}}[\bot, \mathsf{d}_{\mathsf{z}_\mathsf{G}}](\\
&\qquad (t : \mathsf{d}_{\mathsf{z}_\mathsf{G}}) \Rightarrow null, \\
&\qquad (s : \mathsf{d}_{\mathsf{z}_\mathsf{G}}) \Rightarrow s \\
&\quad)
\end{aligned}$$

(e)

$$\begin{aligned}
\mathsf{en}_\mathsf{L}^\mathsf{B}(\mathsf{G},\mathsf{d},\mathsf{e}) &= \mathsf{en}_\mathsf{L}^\mathsf{d}(\mathsf{G},\mathsf{d},\mathsf{e}) \cup \mathsf{en}_\mathsf{L}^\mathsf{b}(\mathsf{G},\mathsf{d},\mathsf{e}) \\
\mathsf{en}_\mathsf{S}^\mathsf{B}(\mathsf{a}^n,\mathsf{G},\mathsf{d},\mathsf{e}) &= \mathsf{en}_\mathsf{S}^\mathsf{b}(\mathsf{a}^n,\mathsf{G},\mathsf{d},\mathsf{e})
\end{aligned}$$

(f)

Figure 7. Implementation Rules

4. Typing MCS

In this section we discuss how to ensure well-typedness of an object language program embedded in MCS. One of the key observations of the previous section is that we can use the tree built to enforce well-formedness of the program. In this section, we explore how to use that to guarantee well-typedness of the program. The key is to use the type checker of the host language to check the type of the object language.

4.1 An Example

First, let's look at an example.

Example 4.1. The simply typed lambda calculus with de Brujin indices, booleans, and integers:

$$\begin{array}{rcl} e & \rightarrow & var(i)|con(n|b)|app\ e\ e|abs\ e \\ i & = & z|s\ i \\ n & = & \text{integers} \\ b & = & \text{booleans} \\ \\ t & = & N|B|fun\ t\ t \\ \\ E & = & nil|\{t, E\} \end{array}$$

where e are terms, t are types, and E are typing environments. And the following typing rules:

$$\frac{}{E \vdash con(n) : N} \quad (\text{TNAT}) \qquad \frac{}{E \vdash con(b) : B} \quad (\text{TBOOL})$$

$$\frac{E \vdash var(i) : t_1}{\{t_2, E\} \vdash var(s\ i) : t_1} \quad (\text{TSUCC})$$

$$\frac{}{\{t, E\} \vdash varx(z) : t} \quad (\text{TZERO})$$

$$\frac{E \vdash e_1 : Fun(t_1, t_2) \qquad E \vdash e_2 : t_1}{E \vdash app\ e_1\ e_2 : t_2} \quad (\text{TAPP})$$

$$\frac{\{t_1, E\} \vdash e : t_2}{E \vdash abs(t_1)\ e : fun(t_1, t_2)} \quad (\text{TABS})$$

where t, t_1, t_2 are type metavariables.

As in the previous example, we look at how to embed both the grammar and the typing as case classes. In the previous example, we used an embedding en^{d} to map terms of the object language to terms of the host language. Here, we also need a mapping that maps types of the object language to the host language. A mapping function ε maps the types of the object langauge to Scala.

$$\begin{array}{rcl} \varepsilon_t(N) & = & Int \\ \varepsilon_t(B) & = & Boolean \\ \varepsilon_t(fun\ \mathsf{t}_1\ \mathsf{t}_2) & = & Fun[\varepsilon_t(\mathsf{t}_1), \varepsilon_t(\mathsf{t}_2)] \end{array}$$

Here we use the metametavariable $\mathsf{t}_1, \mathsf{t}_2$ to disambiguate with type metavariables t_1, t_2. For booleans and integers, we use Scala's types, and for function type, we defined a new type constructor to disambiguate with the function type in Scala.

Let us first focus on the rules (TNAT), (TBOOL), (TAPP), and partial (TABS) without the typing environment. We really can not claim well-typedness without the rest of the rules, but we will consider those rules later. We represent types and terms of the object language using case classes as shown in Figure 8.

Next we implement the embedding classes. Since the grammar is already in GNF, we can encode it directly in Scala, this time parametrize e_e also by χ, the type of the term that is currently being constructed. In a first attempt, we tried to implement it with a builder function as

```
class E[κ, σ, χ](b : Term[χ] ⇒ σ, k : σ ⇒ κ){
  ...
  def abs[χ1, χ2]() : E[κ, σ, χ2] =
      new E((t : Term[χ2]) ⇒ b(Abs[χ1, χ2](t)), k)
}
```

But this program does not compile, because in the definition of function abs the builder b expects some value of type $Term[\chi]$, but what we supplied is of type $Term[Fun[\chi_1, \chi_2]]$. To solve the type mismatch, we need to test the definitional equality of types, which

```
case class Fun[τ1, τ2]()

abstract case class Term[τ]()
case class Var(x : Ref[τ]) extends Term[τ]()
case class Con(x : τ) extends Term[τ]()
case class App[τ1, τ2](t1 : Term[Fun[τ1, τ2]], t2 : Term[τ1])
  extends Term[τ2]()
case class Abs[τ1, τ2](t : Term[τ1])
  extends Term[Fun[τ1, τ2]]()

abstract case class Ref[τ]()
case class Zero[τ]() extends Ref[τ]()
case class Succ[τ](r : Ref[τ]) extends Ref[τ]()
```

Figure 8. Data structure for Implementation with a Builder Function and Typing, Partial Version

```
class E[κ, σ, χ](b : Term[χ] ⇒ σ, k : σ ⇒ κ){
  def varx(i : Ref[χ]) : κ = k(b(Var[χ](i)))
  def con(c : χ) : κ = k(b(Con[χ](c)))
  def app[χ1]() :
    E[E[κ, σ, χ1], Term[χ1] ⇒ σ, Fun[χ1, χ]] =
      new E(
        (t1 : Term[Fun[χ1, χ2]]) ⇒
          (t2 : Term[χ1]) ⇒ b(App(t1, t2)),
        (b1 : Term ⇒ σ) ⇒ E(b1, k))
  def abs[χ1, χ2](cast : Term[χ1, χ2] ⇒ Term[χ]) :
    E[κ, σ, χ2] =
      new E((t : Term[χ2]) ⇒
        b(cast(Abs[χ1, χ2](t))), k)
}
```

Figure 9. Implementation with a Builder Function and Typing, Partial Version

can be done by passing in a cast function: $cast : Term[\chi_1, \chi_2] \Rightarrow Term[\chi]$. The modified code is shown in Figure 9.

We take a simple approach here – we assume that when the method app is called we pass the following generic function

$$\textbf{def}\ id[\tau](x : \tau) = x$$

and initiate the type parameter to the desired type of the abstraction. For example, $abs(id[Fun[Int, Int]])$. Note that if the type is incorrect, then Scala will report type errors, which means that the only "requirement" for the user is ensuring that other functions are not passed in as the cast function. We postpone the discussion of a more sophisticated way of automatically enforcing definitional equality of types that removes this "requirement" to Section 6.

We have now implemented (TNAT), (TBOOL), (TAPP), and partial (TABS). Next, we add typing environments to complete the implementation. First, we define a mapping for typing environments.

$$\begin{array}{rcl} \varepsilon_e(nil) & = & Nothing \\ \varepsilon_e(\{\mathsf{t}, \mathsf{E}\}) & = & (\varepsilon_t(\mathsf{t}), \varepsilon_e(\mathsf{E})) \end{array}$$

We add a type parameter that represents the typing environment to all classes that represent terms, The Ref class and its subclasses represent the typing rules (TZERO), (TSUCC) as in [2]. Now the classes together guarantee that only well-formed and well-typed terms can be constructed. The modificationt to the classes are shown in Figure 10.

We made two changes here. First, we added a new type parameter η. Second, in the type of abs, we modified it to implement the modification of the environment in (TABS). Note that here we only

```
abstract case class Term[τ, η]()
case class Var[τ, η](x : Ref[τ, η]) extends Term[τ, η]()
case class Con[τ, η](x : τ) extends Term[τ, η]()
case class App[τ1, τ2, η](
  t1 : Term[Fun[τ1, τ2], η], t2 : Term[τ1, η])
  extends Term[τ2, η]()
case class Abs[τ1, τ2, η](t : Term[τ2, (τ1, η)])
  extends Term[Fun[τ1, τ2], η]()

abstract case class Ref[τ, η]()
case class Zero[τ, η]() extends Ref[τ, η]()
case class Succ[τ, η](r : Ref[τ, η])
  extends Ref[τ, (_, η)]()

class E[κ, σ, χ, η](b : Term[χ] ⇒ σ, k : σ ⇒ κ){
  def varx(i : Ref[τ, η]) : κ = k(b(Var[χ, η](i)))
  def con(c : χ) : κ = k(b(Con[χ, η](c)))
  def app[χ1, η]() :
    E[E[τ, σ, χ1, η], Term[χ1, η] ⇒ σ, Fun[χ1, χ], η] =
      new E(
        (t1 : Term[Fun[χ1, χ2], η]) ⇒
          (t2 : Term[χ1, η]) ⇒ b(App[χ1, χ, η](t1, t2)),
        (b1 : Term[χ, η] ⇒ σ) ⇒ E(b1, k))
  def abs[χ1, χ2, η](cast : Term[χ1, χ2, η] ⇒ Term[χ, η]) :
    E[κ, σ, χ2, (χ1, η)] =
      new E((t : Term[χ2, (χ1, η)]) ⇒
        b(cast(Abs[χ1, χ2, η](t))), k)
}
```

Figure 10. Implementation with a Builder Function and Typing, Complete Version

need to change the environment in the continuation because, by the typing rule (TABS), the new environment is used only in the term immediately following abs.

We can encode terms in L(G) in Scala. To use this embedding, not only do we need to pass in an initial builder to the E class where we create an object of the class, but also specify the type parameters representing the initial typing environment and the type of the program. We define auxiliary methods.

```
def term[χ]() = new E[
  ⊥,
  Term[χ, Nothing],
  χ,
  Nothing](
  (t : Term[χ, Nothing]) ⇒ null,
  (s : Term[χ, Nothing]) ⇒ s
)
def id[τ, η](x : Term[τ, η]) = x
```

Using these methods, the following term

$$app\ abs\ varx(z)\ con(1)$$

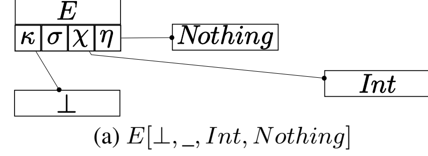
is encoded in Scala as

```
term[Int]() app()
  abs(id[Fun[Int, Int], Nothing]) varx(Zero()) con(1) : ⊥
```

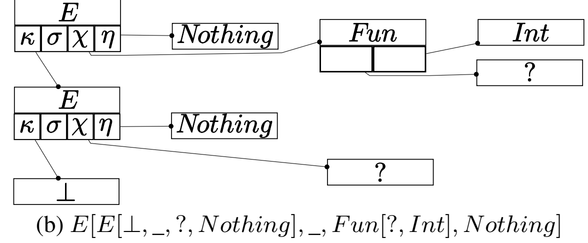
The types of the partial method chains are shown in Figure 11. The type of initial $term()$ is shown in Figure 11(a); The type of $term()$ $app()$ is shown in Figure 11(b); and so forth. We do not care about the type σ, therefore, we put "_". The question marks stand for type variables. The type of the whole method chain is shown in Figure 11(e).

As a more complex example, the following term

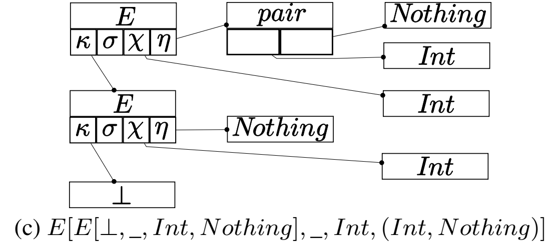
$$app\ app\ abs\ abs\ con(2)\ app\ abs\ varx(z)\ con(1)\ con(1)$$



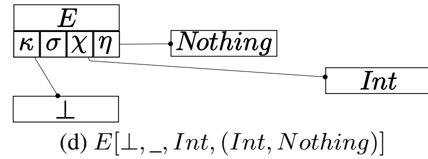
(a) $E[\bot, _, Int, Nothing]$



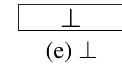
(b) $E[E[\bot, _, ?, Nothing], _, Fun[?, Int], Nothing]$



(c) $E[E[\bot, _, Int, Nothing], _, Int, (Int, Nothing)]$



(d) $E[\bot, _, Int, (Int, Nothing)]$



(e) $\bot$

Figure 11. Examples

is encoded in Scala as

```
term[Int]()
  app()
    app()
      abs(id[Fun[Int, Fun[Int, Int]], Nothing])
        abs(id[Fun[Int, Int], (Int, Nothing)])
          con(2)
      app()
        abs(id[Fun[Int, Int], Nothing])
          varx(Zero())
        con(1) : ⊥
```

4.2 Embedding Typing

Before we proceed to define the general metafunction en^{T}, we would like to summarize what we have learned from the example. First of all, we must take typing rules into account when defining this metafunction. It is very difficult to define a general embedding function for all type systems, because it is very difficult to define a small language that is expressive enough for all type systems. However, if we focus on languages built around simply typed lambda calculus, then there are a few assumptions that we can make. The most important assumption is that

Assumption 4.2. *There is exactly one typing rule for each production in the grammar.*

Each typing rule

$$\frac{\mathsf{E}_{s_1} \vdash \mathsf{e}_{s_1} : \mathsf{t}_{s_1} \quad \dots \quad \mathsf{E}_{s_m} \vdash \mathsf{e}_{s_m} : \mathsf{t}_{s_m}}{\mathsf{E} \vdash \mathsf{ae}_1 \dots \mathsf{e}_n : \mathsf{t}} \quad (\mathcal{TR}^*)$$

can be written as

$$\frac{\begin{array}{c}\mathsf{E}_{\mathsf{a},s_1}(\mathsf{E},\mathsf{t}) \vdash \mathsf{e}_{s_1} : \mathsf{T}_{\mathsf{a},s_1}(\mathsf{t}) \\ \dots \\ \mathsf{E}_{\mathsf{a},s_m}(\mathsf{e},\mathsf{t}) \vdash \mathsf{e}_{s_m} : \mathsf{T}_{\mathsf{a},s_m}(\mathsf{t})\end{array}}{\mathsf{E} \vdash \mathsf{ae}_1 \dots \mathsf{e}_n : \mathsf{t}} \quad (\mathcal{TR})$$

where $\{s_1, \dots, s_m\} \subset \{1, \dots, n\}$. For each typing rule $\mathcal{TR}$, we define two groups of mappings. The first group is $\mathsf{T}_{\mathsf{a},s_i}$, which maps the type t of $\mathsf{e}_1 \dots \mathsf{e}_n$ to the types of $\mathsf{e}_{s_1}, \dots, \mathsf{e}_{s_m}$. Another is $\mathsf{E}_{\mathsf{a},s_i}$ which maps the environment E and type t of $\mathsf{e}_1 \dots \mathsf{e}_n$ to the environments of $\mathsf{e}_{s_1}, \dots, \mathsf{e}_{s_m}$. The second assumption is that

Assumption 4.3. *Both the group $\mathsf{T}_{\mathsf{a},s_i}$ and the group $\mathsf{E}_{\mathsf{a},s_i}$ are definable with either construction or pattern matching on the parameters.*

It turns out that many syntactic sugars fall into this framework.

Example 4.4. For example, consider extending our language with the rules for `if-then-else` expressions and pairs.

$$\frac{E \vdash e_1 : B \quad E \vdash e_2 : t \quad E \vdash e_3 : t}{E \vdash if\ e_1\ then\ e_2\ else\ e_3 : t} \quad (\text{TIfThenElse})$$

$$\frac{E \vdash e_1 : t_1 \quad E \vdash e_2 : t_2}{E \vdash pair\ e_1\ e_2 : (t_1, t_2)} \quad (\text{TPair})$$

$$\frac{E \vdash e : (t_1, t_2)}{E \vdash fst\ e : t_1} \quad (\text{TFst}) \qquad \frac{E \vdash e : (t_1, t_2)}{E \vdash snd\ e : t_2} \quad (\text{TSnd})$$

The (TIfThenElse) rules can be written as

$$\frac{\begin{array}{c}\mathsf{E}_{if,1}(E,t) \vdash e_1 : \mathsf{T}_{if,1}(t) \\ \mathsf{E}_{if,3}(E,t) \vdash e_2 : \mathsf{T}_{if,3}(t) \\ \mathsf{E}_{if,5}(E,t) \vdash e_3 : \mathsf{T}_{if,5}(t)\end{array}}{E \vdash if\ e_1\ then\ e_2\ else\ e_3 : t} \quad (\text{TIfThenElse})$$

where

$$\begin{array}{rcl}\mathsf{E}_{if,1}|\mathsf{E}_{if,3}|\mathsf{E}_{if,5}(\mathsf{E},\mathsf{t}) & = & \mathsf{E} \\ \mathsf{T}_{if,1}(\mathsf{t}) & = & B \\ \mathsf{T}_{if,3}|\mathsf{T}_{if,5}(\mathsf{t}) & = & \mathsf{t}\end{array}$$

The (TPair) rule can be written as

$$\frac{\begin{array}{c}\mathsf{E}_{pair,1}(E,(t_1,t_2)) \vdash e_1 : \mathsf{T}_{pair,1}(t) \\ \mathsf{E}_{pair,2}(E,(t_1,t_2)) \vdash e_2 : \mathsf{T}_{pair,3}(t)\end{array}}{E \vdash pair\ e_1\ e_2 : (t_1, t_2)} \quad (\text{TPair})$$

where

$$\begin{array}{rcl}\mathsf{E}_{pair,1}|\mathsf{E}_{pair,2}(\mathsf{E},\mathsf{t}) & = & \mathsf{E} \\ \mathsf{T}_{pair,1}((\mathsf{t},_)) & = & \mathsf{t} \\ \mathsf{T}_{pair,2}((_,\mathsf{t})) & = & \mathsf{t}\end{array}$$

The (TAbs) rule can be written as

$$\frac{\mathsf{E}_{abs,1}(E, fun(t_1,t_2)) \vdash e : \mathsf{T}_{abs,1}(fun(t_1,t_2))}{E \vdash abse : fun(t_1,t_2)} \quad (\text{TAbs})$$

where

$$\begin{array}{rcl}\mathsf{E}_{abs,1}(\mathsf{E}, fun(\mathsf{t},_)) & = & \{\mathsf{t},\mathsf{E}\} \\ \mathsf{T}_{abs,1}(fun(_,\mathsf{t})) & = & \mathsf{t}\end{array}$$

The (TApp) rule can be written as

$$\frac{\begin{array}{c}\mathsf{E}_{app,1}(E,t) \vdash e_1 : \mathsf{T}_{app,1}(t) \\ \mathsf{E}_{app,2}(E,t) \vdash e_2 : \mathsf{T}_{app,2}(t)\end{array}}{E \vdash e_1\ e_2 : t} \quad (\text{TApp})$$

where

$$\begin{array}{rcl}\mathsf{E}_{app,1}|\mathsf{E}_{app,2}(\mathsf{E},\mathsf{t}) & = & \mathsf{E} \\ \mathsf{T}_{app,1}(\mathsf{t}) & = & fun(t_1,\mathsf{t}) \\ \mathsf{T}_{app,2}(\mathsf{t}) & = & t_1\end{array}$$

where t_1 is a fresh type metavariable.

First, if type string matching or environment string pattern matching is required in the consequence of a typing rule for $\mathsf{z} \rightarrow \mathsf{az}_1 \dots \mathsf{z}_n$ (which means that the type of the consequence is not a type metavariable), such as in (TAbs) and (TPair), then we need to pass in a proof of definitional equality by a *cast* function.

Second, if type string pattern matching or environment string pattern matching is required in the antecedent of a typing rule for $\mathsf{z} \rightarrow \mathsf{az}_1 \dots \mathsf{z}_n$ (which means that construction is required in $\mathsf{T}_{\mathsf{a},k}$ or $\mathsf{E}_{\mathsf{a},k}$, for some $k \in \{1, \dots, n\}$), then we need to construct a new type for parameter χ or η, which represents the type and the environment, respectively, either for type constructors that are pushed onto the continuation stack in the return type of e_a or the types of the parameters of e_a.

Third, all type variables in the rules other than that represented by χ become the type variables of the method definitions such as the *app* and *abs* method.

By viewing the grammar as parametrized, we actually embedded part of the grammar not in the method chaining style but in the functional nesting style. Therefore, it is possible to use some of the existing techniques to avoid requiring the *cast* function, such as in (TSucc). Conversely, because an initial object with necessary information is needed to initialize a method chain, the method chaining style would induce unnecessary overhead and clog to the code if used within the functional nesting style. This implies that we can use method chaining style to embed a language while using functional nesting style to deal with variables.

One potential problem that scrutinous readers may have noticed is that in Example 4.1, the typing rules for constants are only partially represented because the *con* method does not receive a cast function as an argument. As a result, the universe of types may be expanded. In general, the cast function is necessary. However, this is not a problem in our example since all types of constants are either specified in the type argument of *term*, or by the type argument of cast functions for *abs*. Even if we add a cast function as a parameter of the *con* method, we still rely on the user to provide a type argument within the defined types for the cast function. We call these types that are reused in the object language "native" types. We need to deal with these types carefully, when generalizing our method examplified.

4.3 The General Embedding Function for Typed Languages

Now, let's define the embedding en^T. en^T is defined by a quadruple of embedding functions $(\mathsf{en}^\mathsf{t}, \mathsf{en}^\mathsf{e}, \mathsf{en}^\mathsf{d}, \mathsf{en}^\mathsf{b})$. These embedding functions are parametrized over G, T, E, R, e, and d. G is a parametrized grammar of the object language in LL(1) GNF. T is a grammar for types of the object language in LL(1) GNF such that $\mathsf{Z}_\mathsf{T} = \{\mathsf{z}_\mathsf{T}\}$ (which may be a subgrammar of G). E is a grammar for typing environments of the object language in LL(1) GNF such that $\mathsf{Z}_\mathsf{E} = \{\mathsf{z}_\mathsf{E}\}$. R is a function that maps each production in G to a typing rule of the form $\mathcal{TR}$, so that there is exactly one typing rule for each production in G. The name embedding function e maps terminals and nonterminals in G to unique Scala names, and the name embedding function d maps terminals and nonterminals of

$$\mathsf{en}^{\mathsf{t}}_{\mathsf{L}}(\mathsf{T},\mathsf{d}) = \{\mathsf{en}^{\mathsf{t}}_{\mathsf{L}}(\mathsf{p},\mathsf{d})|\mathsf{p}\in\mathsf{P}_{\mathsf{T}}\}$$
$$\mathsf{en}^{\mathsf{t}}_{\mathsf{L}}(\mathsf{z}\rightarrow\mathsf{az}^n,\mathsf{d}) = \textbf{case class } \mathsf{d}_{\mathsf{a}}[v_1,\ldots,v_n]$$

(a)

$$\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{a}^n,\mathsf{d}) = \mathsf{en}^{\mathsf{t}}_{\mathsf{S}}{}'(\mathsf{tree}(\mathsf{a}^n),\mathsf{d})$$
$$\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}{}'(\mathsf{z}(\mathsf{at}^n),\mathsf{d}) = \mathsf{d}_{\mathsf{a}}[\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}{}'(\mathsf{t}^n_1,\mathsf{d}),\ldots,\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}{}'(\mathsf{t}^n_n,\mathsf{d})]$$

(b)

Figure 12. Embedding of Types in the Object Language

$$\mathsf{en}^{\mathsf{e}}_{\mathsf{L}}(\mathsf{T},\mathsf{d}) = \{\mathsf{en}^{\mathsf{e}}_{\mathsf{L}}(\mathsf{p},\mathsf{d})|\mathsf{p}\in\mathsf{P}_{\mathsf{E}}\}$$
$$\mathsf{en}^{\mathsf{e}}_{\mathsf{L}}(\mathsf{z}\rightarrow\mathsf{az}^n,\mathsf{d}) = \textbf{case class } \mathsf{d}_{\mathsf{a}}[v_1,\ldots,v_n]$$

(a)

$$\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}(\mathsf{a}^n,\mathsf{d}) = \mathsf{en}^{\mathsf{e}}_{\mathsf{S}}{}'(\mathsf{tree}(\mathsf{a}^n),\mathsf{d})$$
$$\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}{}'(\mathsf{z}(\mathsf{at}^n),\mathsf{d}) = \mathsf{d}_{\mathsf{a}}[\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}{}'(\mathsf{t}^n_1,\mathsf{d}),\ldots,\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}{}'(\mathsf{t}^n_n,\mathsf{d})]$$

(b)

Figure 13. Embedding of Typing Environments in the Object Language

G, T, and E, and metavariables to unique Scala names such that $\mathsf{d_z} \neq \mathsf{e_z}$ for all $\mathsf{z}\in\mathsf{Z_G}$.

We write the embedding of metavariable t as d_t.

Definition 4.5. A extension $\widetilde{\mathsf{G}}$ of grammar G with metavariables $t_1, t_2, \ldots$ is $(\mathsf{Z_G}, \mathsf{z_G}, \mathsf{A_G}\cup\{t_1,t_2,\ldots\}, \mathsf{P_G}\cup\{\mathsf{z}\rightarrow t_k|\mathsf{z}\in\mathsf{Z_G}, k\in\{1,2,\ldots\}\})$.

We start by defining the embedding en^{t}, which is shown in Figure 12, where $v_1,\ldots,v_n$ are type parameters, a^n is a string in $\widetilde{\mathsf{T}}$, and t^n are subtrees. Each terminal in T is mapped to a type constructor. This embedding generates the Scala classes that represent types in the object language. This embedding actually allows any type of the host language to be a type of the object language. We do not need to restrict those types here because possible types are restricted by the typing rules R. One advantage of this is that we avoid casting for many types that we can reuse from the host language.

The string embedding $\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}$ is defined on $\widetilde{\mathsf{T}}$ which maps type metavariables t to d_t. For example,

$$\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(N,\mathsf{d}) = Int$$
$$\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(B,\mathsf{d}) = Boolean$$
$$\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(fun\ \mathsf{t}_1\ \mathsf{t}_2,\mathsf{d}) = Fun[\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{t}_1,\mathsf{d}),\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{t}_2,\mathsf{d})]$$
$$\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(t,\mathsf{d}) = \mathsf{d}_t$$

The embedding en^{e} of E is shown in Figure 13, where $v_1,\ldots,v_n$ are type parameters, a^n is a string in $\widetilde{\mathsf{E}}$, and t^n are subtrees. Because E usually contains T, the embedding of E may share the same classes as the embedding of T. For example, if the environment is $\{\mathsf{t}_1,\ldots\{\mathsf{t_n},nil\}\}$, then $\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}(\{\mathsf{t}_1,\ldots\{\mathsf{t_n},nil\}\},\mathsf{d}) = \mathsf{d}_{\{}[\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{t}_1,\mathsf{d}),\mathsf{d}_{\{}[\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{t_{n-1}},\mathsf{d}),\mathsf{d}_{nil}]]$.

The typing rules in R are encoded by both en^{d} and en^{b}.

As the name suggests, en^{d} is an extension of the embedding en^{d} defined in Section 3 for productions in G. Each embedding class is parametrized over type parameters χ,η which represent types and typing environments of the object language, respectively. The fv metavariables are replaced by lists of free type variables that occur

$$\mathsf{nat}(\mathsf{a},\mathsf{b},\mathsf{c}) = \begin{cases}\mathsf{b} & \mathsf{b}\text{ native}\\ \mathsf{a}[\mathsf{b},\mathsf{c}] & \text{otherwise}\end{cases}$$
$$\varphi \triangleq \mathsf{nat}(\mathsf{d_z},\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{t},\mathsf{d}),\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}(\mathsf{E},\mathsf{d}))$$
$$\varphi^n_i \triangleq \mathsf{nat}(\mathsf{d}_{\mathsf{z}^n_i},\mathsf{en}^{\mathsf{t}}_{\mathsf{S}}(\mathsf{t}_i,\mathsf{d}),\mathsf{en}^{\mathsf{e}}_{\mathsf{S}}(\mathsf{E}_i,\mathsf{d}))$$

(a)

$$\mathsf{en}^{\mathsf{d}}_{\mathsf{L}}(\mathsf{G},\mathsf{R},\mathsf{d}) = \{\mathsf{en}^{\mathsf{d}}_{\mathsf{L}}(\mathsf{p})|\mathsf{p}\in\mathsf{P_G}\}\cup\{\mathsf{en}^{\mathsf{d}}_{\mathsf{L}}(\mathsf{z})|\mathsf{z}\in\mathsf{Z_G}\}$$
$$\mathsf{en}^{\mathsf{d}}_{\mathsf{L}}(\mathsf{z}\rightarrow\mathsf{az}^n,\mathsf{R},\mathsf{d}) = \textbf{case class } \mathsf{d_a}[\mathsf{fv}](i_1:\varphi^n_1,\ldots,i_n:\varphi^n_n)\ \textbf{extends}\ \varphi()$$
$$\text{where } \mathsf{R}(\mathsf{z}\rightarrow\mathsf{az}^n) = \frac{\mathsf{E}_1\vdash e_1:\mathsf{t}_1 \quad \ldots \quad \mathsf{E}_n\vdash e_n:\mathsf{t}_n}{\mathsf{E}\vdash \mathsf{a}e_1\ldots e_n:\mathsf{t}}$$
$$\mathsf{en}^{\mathsf{d}}_{\mathsf{L}}(\mathsf{z},\mathsf{d}) = \textbf{abstract case class } \mathsf{d_z}[\chi,\eta]()$$

(b)

$$\mathsf{en}^{\mathsf{d}}_{\mathsf{S}}(\mathsf{a}^n,\mathsf{d}) = \mathsf{en}^{\mathsf{d}}_{\mathsf{S}}{}'(\mathsf{tree}(\mathsf{a}^n),\mathsf{d})$$
$$\mathsf{en}^{\mathsf{d}}_{\mathsf{S}}{}'(\mathsf{z}(\mathsf{at}^n),\mathsf{d}) = \begin{cases}\mathsf{d_a} & \mathsf{d_a}\text{ native}\\ \mathsf{d_a}(\mathsf{en}^{\mathsf{d}}_{\mathsf{S}}{}'(\mathsf{t}^n_1,\mathsf{d}),\ldots\mathsf{en}^{\mathsf{d}}_{\mathsf{S}}{}'(\mathsf{t}^n_n,\mathsf{d})) & \text{otherwise}\end{cases}$$

(c)

Figure 14. Embedding of Terms in the Object Language

in the constructor definition. Each constructor represents a typing rule. The techniques for defining these data structures are typing rule directed, as in [2]. Using auxilary definitions in Figure 14(a), where "native" means that the type is a predefined Scala type such as Int, or the name is a name for constants such as integers, the embedding is defined in Figure 14(b) and Figure 14(c), where a^n is a string in G, and t^n are subtrees. Note that the string embedding is exactly the same as in Section 3, because Scala automatically infers the type parameters.

Next, we define $\mathsf{en}^{\mathsf{b}}_{\mathsf{L}}$. The rules are shown in Figure 15. We redefined the auxiliary symbols from Section 3, adding type variables that represent types and typing environments to $\kappa^n_k,\mathsf{B}^{n,p}_k$ for $k,n,p\in\mathbb{N}$. Also, in the definition of $\mathsf{B}^{n,p}_0$, we apply the $cast$ function after applying the builder.

Each method defined in class $\mathsf{e_z}$ corresponds to a production, and its typing rule. The class $\mathsf{e_z}$ is parametrized over κ,σ,χ,η, which represents the continuation stack, the type of the builder, the type of the expected string that is generated by z, and the typing environment, respectively. In the embedding, a $cast:\varphi\Rightarrow\mathsf{d_z}[\mathsf{tvt},\mathsf{tve}]$ function is passed in as a parameter for each method. In the typing rule, the metametavariables t and t_k for $k\in\{1,\ldots,n\}$ are replaced with the actual types and similarly for E and E_k for $k\in\{1,\ldots,n\}$. If the actual type is a type metavariable t and the actual environment is a metavariable E, then $\mathsf{t}=t,\mathsf{E}=E$ and neither $\mathsf{T}_{\mathsf{a}_i,k}$ nor $\mathsf{E}_{\mathsf{a}_i,k}$ require pattern matching, in which case we can ignore the $cast$ function because it has a trivial type $\mathsf{d_z}[\mathsf{d}_t,\mathsf{d_e}]\Rightarrow\mathsf{d_z}[\mathsf{d}_t,\mathsf{d_e}]$, and can be removed in a code generator[3]; but we still list it in the parameters for uniformity. Similarly for $\mathsf{en}^{\mathsf{b}}_{\mathsf{S}}$ defined in Figure 16, the code generator can remove the id argument for trivial cast function parameters to generate more succinct code.

[3] EriLex does this by testing the defintional equality when generating code.

$$
\begin{array}{rcl}
p &=& \mathsf{arity}(\mathsf{z} \to \mathsf{a}\mathsf{z}^p\mathsf{z}^n) \\
\mathsf{nat}(\mathsf{a},\mathsf{b},\mathsf{c}) &=& \begin{cases} \mathsf{b} & \mathsf{b} \text{ native} \\ \mathsf{a}[\mathsf{b},\mathsf{c}] & \text{otherwise} \end{cases} \\
\varphi &\triangleq& \mathsf{nat}(\mathsf{d_z}, \mathsf{en^t_S}(\mathsf{t},\mathsf{d}), \mathsf{en^e_S}(\mathsf{E},\mathsf{d})) \\
\varphi^n_i &\triangleq& \mathsf{nat}(\mathsf{d}_{\mathsf{z}^n_i}, \mathsf{en^t_S}(\mathsf{t}^n_i,\mathsf{d}), \mathsf{en^e_S}(\mathsf{E}^n_i,\mathsf{d})) \\
\varphi^p_i &\triangleq& \mathsf{nat}(\mathsf{d}_{\mathsf{z}^p_i}, \mathsf{en^t_S}(\mathsf{t}^p_i,\mathsf{d}), \mathsf{en^e_S}(\mathsf{E}^p_i,\mathsf{d})) \\
\sigma^n_0 &\triangleq& \sigma \\
\sigma^n_{i+1} &\triangleq& \varphi^n_{n-i} \Rightarrow \sigma^n_i \\
\kappa^n_0 &\triangleq& \kappa \\
\kappa^n_{i+1} &\triangleq& \mathsf{e}_{\mathsf{z}^n_{n-i}}[\kappa^n_i, \sigma^n_i, \mathsf{en^t_S}(\mathsf{t}^n_{n-i},\mathsf{d}), \mathsf{en^e_S}(\mathsf{E}^n_{n-i},\mathsf{d})] \\
\mathsf{k}^n_0 &\triangleq& k \\
\mathsf{k}^n_{i+1} &\triangleq& (b_{i+1} : \sigma^n_{i+1}) \Rightarrow \textbf{new } \mathsf{e}_{z^n_{n-i}}(b_{i+1}, \mathsf{k}^n_i) \\
\mathsf{B}^{n,p}_0 &\triangleq& b(cast(\mathsf{d_a}(i_1,\ldots,i_p,t_1,\ldots,t_n))) \\
\mathsf{B}^{n,p}_{i+1} &\triangleq& (t_{n-i} : \varphi^n_{n-i}) \Rightarrow \mathsf{B}^{n,p}_i
\end{array}
$$

(a)

$$
\begin{array}{rcl}
\mathsf{en^b_L}(\mathsf{G},\mathsf{R},\mathsf{d},\mathsf{e}) &=& \{\mathsf{en^b_L}(\mathsf{p},\mathsf{R},\mathsf{d},\mathsf{e}) | \mathsf{p} \in \mathsf{P}_{\mathsf{G}_1}\} \\
\mathsf{en^b_L}(\mathsf{z} \to \mathsf{a}\mathsf{z}^p\mathsf{z}^n, \mathsf{R},\mathsf{d},\mathsf{e}) &=& \textbf{class } \mathsf{e_z}[\kappa,\sigma,\mathsf{tvt},\mathsf{tve}](\\
&& \quad b : \mathsf{d_z}[\mathsf{tvt},\mathsf{tve}] \Rightarrow \sigma, \\
&& \quad k : \sigma \Rightarrow \kappa)\{\mathsf{en^b_L}(\mathsf{a}\mathsf{z}^n,\mathsf{d},\mathsf{e})\} \\
&& \text{where } \mathsf{R}(\mathsf{z} \to \mathsf{a}\mathsf{z}^n) = \\
&& \dfrac{\mathsf{E}^p_1 \vdash \mathsf{e}^p_1 : \mathsf{t}^p_1 \quad \ldots \quad \mathsf{E}^n_n \vdash \mathsf{e}^n_n : \mathsf{t}^n_n}{\mathsf{E} \vdash \mathsf{a}\mathsf{e}^p\mathsf{e}^n : \mathsf{t}} \\
\mathsf{en^b_L}(\mathsf{a}\mathsf{z}^p\mathsf{z}^n,\mathsf{d},\mathsf{e}) &=& \textbf{def } \mathsf{e_a}[\mathsf{fv}](\\
&& \quad i_1 : \varphi^p_1, \ldots, i_p : \varphi^p_p, \\
&& \quad cast : \varphi \Rightarrow \mathsf{d_z}[\mathsf{tvt},\mathsf{tve}]) : \\
&& \quad \kappa^n_n = \mathsf{k}^n_n(\mathsf{B}^{n,p}_n)
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathsf{tvt} &=& \begin{cases} \chi & \mathsf{t} \text{ is not a metavariable} \\ \mathsf{d_t} & \text{otherwise} \end{cases} \\
\mathsf{tve} &=& \begin{cases} \eta & \mathsf{E} \text{ is not a metavariable} \\ \mathsf{d_E} & \text{otherwise} \end{cases} \\
\mathsf{fv} &=& \text{free type variables in the method definition}
\end{array}
$$

(b)

Figure 15. Embedding of Terms in MCS with a Builder Function and Typing

The source embedding of programs in the object language defined using the following auxiliary method definitions: $term()$ and $id()$, is shown in Figure 16.

The embedding in $\mathsf{en^b_S}$ for a program

$$\mathsf{a}_1\mathsf{i}^{k_1}\ldots\mathsf{a}_n\mathsf{i}^{k_n}$$

where a_k are top level symbols for $k \in \{1,\ldots,n\}$ and i^{k_i} are second level strings for $i \in \{1,\ldots,n\}$, is defined in Figure 16, where $\mathsf{t}^n_k, \mathsf{E}^n_k$ are the types and typing environments of the substrings of program at a_k, for $k \in \{1,\ldots,n\}$.

The embedding of the object language is defined in Figure 17.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathsf{en^b_S}(\mathsf{a}_1\mathsf{i}^{k_1}\ldots\mathsf{a}_n\mathsf{i}^{k_n},} \\
\mathsf{G},\mathsf{R},\mathsf{d},\mathsf{e},\mathsf{t}^n,\mathsf{E}^n) &=& \textbf{new } term()\mathsf{s}_1\ldots\mathsf{s}_n : \bot \\
&& \text{where} \\
\mathsf{s}_i &=& \mathsf{e}_{\mathsf{a}_i}(\mathsf{en^d_S}(\mathsf{i}^{k_i}_1,\mathsf{d}),\ldots,\mathsf{en^d_S}(\mathsf{i}^{k_i}_{k_i},\mathsf{d}) \\
&& \quad id[\mathsf{en^t_S}(\mathsf{t}^n_i,\mathsf{d}),\mathsf{en^e_S}(\mathsf{E}^n_i,\mathsf{d})]) \\
\varphi_\mathsf{G} &\triangleq& \mathsf{d}_{\mathsf{z_G}}[\mathsf{en^t_S}(\mathsf{t}^n_1,\mathsf{d}),\mathsf{en^e_S}(\mathsf{E}^n_1,\mathsf{d})] \\
\textbf{def } term() &=& \textbf{new } \mathsf{e}_{\mathsf{z_G}}[\bot,\varphi_\mathsf{G},\mathsf{en^t_S}(\mathsf{t}^n_1,\mathsf{d}),\mathsf{en^e_S}(\mathsf{E}^n_1,\mathsf{d})](\\
&& \quad (t : \varphi_\mathsf{G}) \Rightarrow null, \\
&& \quad (s : \varphi_\mathsf{G}) \Rightarrow s \\
&&) \\
\textbf{def } id[\tau,\eta] &=& (x : Term[\tau,\eta]) \Rightarrow x
\end{array}
$$

Figure 16. String Embedding

$$
\begin{array}{rcl}
\mathsf{en^T_L}(\mathsf{T},\mathsf{E},\mathsf{G},\mathsf{R},\mathsf{d},\mathsf{e}) &=& \mathsf{en^t_L}(\mathsf{T},\mathsf{d}) \cup \mathsf{en^e_L}(\mathsf{E},\mathsf{d}) \\
&& \cup \mathsf{en^d_L}(\mathsf{G},\mathsf{R},\mathsf{d}) \cup \mathsf{en^b_L}(\mathsf{G},\mathsf{R},\mathsf{d},\mathsf{e}) \\
\mathsf{en^T_S}(\mathsf{a}^n,\mathsf{G},\mathsf{d},\mathsf{e},\mathsf{t}^n,\mathsf{E}^n) &=& \mathsf{en^b_S}(\mathsf{a}^n,\mathsf{G},\mathsf{d},\mathsf{e},\mathsf{t}^n,\mathsf{E}^n)
\end{array}
$$

Figure 17. The Embedding $\mathsf{en^T_L}$

We need the following theorem to complete our discussion.

Theorem 4.6. *If* G, T, *and* E *are grammars of the terms, types, and typing environments of the object language in LL(1) GNF,* R *is a function that maps each production in* G *to a typing rule of the form* $\mathcal{TR}$, e *maps* $\mathsf{A_G} \cup \mathsf{Z_G}$ *to unique Scala names,* d *maps* $\mathsf{A}_{\tilde{\mathsf{T}}} \cup \mathsf{Z}_{\tilde{\mathsf{T}}} \cup \mathsf{A}_{\tilde{\mathsf{E}}} \cup \mathsf{Z}_{\tilde{\mathsf{E}}} \cup \mathsf{A_G} \cup \mathsf{Z_G}$ *to unique Scala names such that* $\mathsf{d_z} \neq \mathsf{e_z}$ *for all* $\mathsf{z} \in \mathsf{Z_G}$, $\mathsf{s} = \mathsf{a}_1\mathsf{i}^{k_1}\ldots\mathsf{a}_n\mathsf{i}^{k_n} \in \mathsf{A_G}^*$, t^n *are strings of* T,E^n *are strings of* E*, then* $\mathsf{s} \in \mathsf{L}(\mathsf{G})$ *and* $\mathsf{E}^n_k \vdash \mathsf{s}_k : \mathsf{t}^n_k$*, where* s_k *is the substring of* s *at* a_k*, for all* $k \in \{1,...,n\}$ *if and only if* $\mathsf{en^T_L}(\mathsf{T},\mathsf{E},\mathsf{G},\mathsf{R},\mathsf{d},\mathsf{e}) \cup \{\mathsf{en^T_S}(\mathsf{s},\mathsf{G},\mathsf{d},\mathsf{e},\mathsf{t}^n,\mathsf{E}^n)\}$ *is well-typed in Scala.*

Proof. (Sketch) First prove that any encoded term is automatically well-formed and well-typed because the constructor $\mathsf{d_a}$ only encodes well-formed and well-typed terms. Next, by type safety, the encoding always constructs encoding of some well-formed and well-typed term. Finally, we prove that the term constructed is the same term as that is encoded in the method chain. □

It is clear that with the addition of any nontrivial typing (i.e. everything does not have the same type), our embedding generally can no longer be modeled by an slrpda with finite stack symbols and input symbols. However, if we allow the slrpda to have countably infinite number of stack symbols and input symbols, our embedding can be easily modeled, by viewing types constructed from embedding classes as just a complex stack symbol, thus providing an upper bound of the expressiveness of the embedding framework.

5. Related Work

The problems of embedding a typed DSL into a host language in the FNS has been studied extencsively, see [6] for a list of references. However, MCS embedding seems to be largely ignored, probably because most of the research are done in a functional setting, and embedding MCS in a functional programming language without object-oriented features is awkward. As the object-oriented programming languages become more and more mature in their type

systems, MCS may be a viable alternative to FNS since it seems to be very natural in object-oriented programming languages.

Scala[1, 4, 9, 12] is well known for its combination of flexible syntax, functional programming features, and object-oriented programming features, and consequently the ease to define DSLs. The Scala Parser Combinators is a parser library that embeds a BNF-like parser DSLs in the Scala programming language in a very succinct way. Combinator based parser libraries are also widely implemented and used[5]. We would like to point out that the monadic parsers provide a very concise way of embedding parser DSLs. However, as pointed out in [3], there is no easy way to transform the resulting parser to make optimizations. In our approach, the builder function are used to build a typed abstract syntax tree so that the techniques introduced in [3] can be applied. Also, syntactically, which is probably less essential, in combinator based parser libraries, the sequencing combinator (which usually does not appear in BNF) usually can not be omitted unless there are special language support in the host language (such as the do notation in Haskell), whereas in our approach as shown in Section 2, the sequencing operator is not needed, which results in a more succinct embedding. Also, it is not clear in an object-oriented programming language how the advantages of parser combinator would translate to embedding DSLs other than parser DSLs, such as simply typed lambda calculus based languages.

Two case studies of type safe DSLs and many practical aspects of implementing embedded DSL are discussed in [7], in which type safety is dealt with in a case by case manner. Typing environments are partially encoded in class names which results in that the number of classes required is linear to the maximum size of the typing environment allowed. In our approach, this problem is solved based on the embedding from [2] and using the two groups of functions E, T defined in Section 4.2. In this respect, our embedding can be viewed as an extension of embedding from [2].

6. Discussion

It may seem that the difference between FNS and MCS are purely syntactical because one can be encoded in another, but we believe that MCS is a very important tool for object-oriented programming languages. On one hand, the syntax is an important aspect of embedding a DSL, for the expressiveness of an embedded language can not exceed that of the host language. On the other hand, if encoding is concerned only, then most common features in object-oriented programming languages can be encoded in a functional programming language as demonstrated in [10]. Conversely, functional programming can also be encoded in a object-oriented programming language with parametric types and closure. But alternatives are still useful because some encoding may make the code "bulky" at best, and inefficient at worst. After all every programming language is more polished at the task it is intended for. For mainstream object-oriented programming languages, it is creating objects and invoking methods, but not dealing with higher order functions.

Our approach may not be the most efficient because the builder function in the embedding actually builds an AST. However, because the constructors $\mathsf{d_a}$ are not required to be constructors, they can be any functions with the same type. Therefore, they can be easily interchanged with functions that actually act as the interpreter, thus improving the performance. Practically, if we want to obtain the AST for transformation, we can simply change the initial continuation k to $(t : Term) \Rightarrow t$ to make it return the tree built instead of $null$. Sometimes if the grammar is complex, the resulting AST of GNF may not be the desired AST. For example in Example 2.10, we may discard the nodes generated by r' and move its subnodes one level up, as it is apparent that only nodes generated by r are needed in a more compact form of the grammar.

To ensure that the cast function is always the identity function, we may use the technique in [2], but this requires higher-rank polymorphism of kind $* \rightarrow *$ (i.e. universally quantified type constructor in the domain type of a function), which is not directly supported but can be encoded in Scala. For practical reasons we choose not to used this encoding in this framework. However, if direct support for higher-rank polymorphism of higher kind is implemented in Scala, then we will definitely use that because that seems to be a more elegant solution.

Our framework actually allows the object language to have not just "terms", but multiple classes of programming constructs, which is useful for embedding practical DSLs such as SQL in which different classes of programming constructs are not interchangeable.

To conclude, we demonstrated how to embed DSLs in Scala in this paper in a general framework. We tried to keep to the object-oriented programming paradigm as close as possible, so that our framework can be used in a wide range of object-oriented programming languages and are easily assessable to the majority of programmers. In fact, most of the framework can be directly translated to Java, albeit in a more verbose manner, using the classical inner class encoding. There are many other features in Scala that we have not used, such as abstract types, existential types, and explicit bounded polymophism. Exploring these features for embedding DSLs is part of future work for seeking more elegant solutions.

References

[1] K.U.Leuven Adriaan Moors, Frank Piessens and Martin Odersky. Safe type-level abstraction in scala. In *FOOL '08: International Workshop on Foundations of Object-Oriented Languages*, 2008.

[2] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166, New York, NY, USA, 2002. ACM.

[3] Arthur I. Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 15–26, New York, NY, USA, 2009. ACM.

[4] Vincent Cremet, Francois Garillot, Serguei Lenglet, Martin Odersky, Ecole Normale Superieure, and Ecole Normale Superieure De Lyon. A core calculus for scala type checking. In *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science, Springer LNCS*, pages 1–23. Springer, 2006.

[5] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[6] Oleg Kiselyov Jacques Carette and Chung-Cheih Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19:509–543, September 2009.

[7] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for java. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 189–197, New York, NY, USA, 2008. ACM.

[8] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *LINQ in Action*. February 2008.

[9] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 423–438, New York, NY, USA, 2008. ACM.

[10] Benjamin C. Pierce. *Types and Programming Languages*. 2002.

[11] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 1: word, language, grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[12] Dean Wampler and Alex Payne. *Programming Scala*. 2009.