# EriLex: An Embedded Domain Specific Language Generator

Hao Xu

University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
xuh@cs.unc.edu

**Abstract.** EriLex is a software tool for generating support code for embedded domain specific languages (EDSL). EriLex supports defining the syntax, static semantics, and dynamic semantics of EDSLs desigined in the method chaining style, the functional nesting style, or both. EriLex supports various features of EDSLs that are commonly used in manually written EDSL libraries, in addition to other less frequently used features such as higher-order functions and simple types.

## 1 Introduction

An object-oriented (OO) software library usually provides an application programming interface (API) which consists of elements such as objects, classes, and methods that are related to the functionality of the library. There are two roles involved in a software library, library writers and library users. Library writers program the source code of the software library; library users use the API and functionality provided by the library to write other programs.

Sometimes the API of a software library is designed in the Method Chaining Style (MCS) so that consecutive method calls can be chained together according to a set of rules. The MCS style of programming is used in several well-known software libraries, such as jMock[1] and Hibernate Criteria Query[8]. When using software libraries with API in the MCS, the method chains can be viewed as programs written in an embedded domain specific language (EDSL) of which the building blocks are method calls.

One of the advantages of MCS EDSLs in software library design is that it helps grouping logically related method calls into one compact piece of code, as illustrated by the following example in Java.

*Example 1.* An example of Hibernate Criteria Query EDSL program.[8]

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

However, manually coding MCS EDSLs for software libraries that support complex method chains and composition rules can be very tedious and error-prone and few software tools are available that allow library writers to specify these rules in a declarative language from which source code for the software library can be automatically generated.

The main motivation of creating EriLex is to provide such a tool. EriLex supports features of common EDSLs as shown in Example 1 in addition to other features such as types and abstract syntax tree (AST) builders. Also, EriLex is designed to support multiple code generation targets (host languages); currently, the available code generation targets are Java, which will be used in this paper, and (experimentally) Scala.

This paper is organized as follows: Section 2 briefly overviews the tool. Section 3 presents features of the EriLex Specification Language using a few examples; Section 4 informally describes how the EriLex Code Generator works; Section 5 discusses tool reuse and improving the usability of generated EDSLs. Section 6 discusses related work. Section 7 summarized the paper.

## 2 EriLex Overview

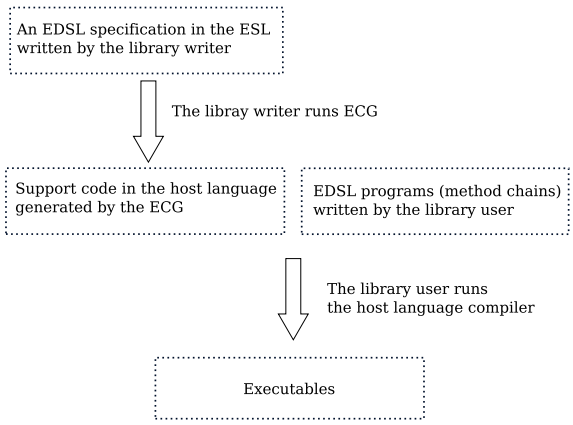



**Fig. 1.** EriLex Workflow

EriLex is composed of two main components, the EriLex Specification Language (ESL) and the EriLex Code Generator (ECG).

The high-level workflow of EriLex is illustrated in Figure 1. The library writer specifies the rules and components, including the syntax and semantics, of an MCS EDSL in a specification in the ESL. Then the library writer selects a host language and runs the ECG which generates from the specification the source code for the software library, called the support code, that implements these rules and components as classes and methods in the host language, so that a

library user can write EDSL programs (as method chains of the host language) according to the specification. To run an EDSL program that a library user writes, the library user only needs to compile the EDSL program and the support code using the host language compiler and run the compiled program using the host language runtime.

For every EDSL specification, the ECG generates two methods: `prog` and `run`. `prog` is used to start a EDSL program (method chain), and `run` is called at the end of the EDSL program to run it. When the library user executes a chain of methods that starts with `prog` and ends in `run`, the generated support code builds up an AST of the EDSL program, and when the `run` method is executed, the AST is evaluated according to the semantics defined in the specification of the EDSL. In the next section, we discuss how to specify the syntax and semantics of an EDSL in the ESL.

## 3 EriLex Specification Language

### 3.1 A Basic EDSL Example

A minimal specification of an EDSL is composed of two sections for syntax and dynamic semantics, respectively.

*Example 2.* Natural numbers.

```
1  syntax
2  e -> zero
3  e -> succ e
4
5  dynamic
6  evaluate: Integer
7  e -> zero {
8      return 0;
9  }
10 e -> succ e {
11     return evaluate(e)+1;
12 }
```

The syntax section (Line 1 to Line 3) starts with `syntax` and consists of a definition of an LL(1) context-free grammar. Each line in this section of the form $nt -> t\ nt_1\ \ldots\ nt_n$ defines a production of the grammar, where $nt, nt_k$ are nonterminals for $k \in \{1, \ldots, n\}$ and $t$ is a terminal[1]. In the support code and EDSL programs, the production define a method $t$ and the context-free grammar defines how the defined methods can be composed to form valid method chains.

The dynamic semantics section (Line 5 to Line 12) starts with `dynamic` and consists of definitions of evaluators. A definition of an evaluator starts with a line of the form $eval : htype$, where $eval$ is the name of the evaluator and $htype$ is the return type of the evaluator, following which are definitions of the form

[1] This is also called the Greibach Normal Form.

$P$ { host language code } which defines a component of the evaluator for terms produced by production $P$. In the host language code, the nonterminals on the right hand side of the production can be used as variable and *eval* can be used as a method.

Running the ECG on this specification produces several Java classes, one of which is the `Util` class which has the `prog` method which is a class method. For the nonterminal `e`, a class `Ee` is generated where `E` is the default prefix for generated classes for nonterminals. The `Ee` class has the `zero`, `succ`, and `run` methods. An example of programs in this EDSL is `prog().succ().succ().succ().succ().zero().run()`.

### 3.2 A Simple Typed EDSL

*Example 3.* Adding booleans and conditional expression to the EDSL in Example 2.

```
1  syntax
2  e -> zero
3  e -> succ e
4  e -> true
5  e -> false
6  e -> if e then e else e
7
8  dynamic
9  evaluate: Object
10 e -> zero {
11     return 0;
12 }
13 e -> succ e {
14     return (Integer)evaluate(e)+1;
15 }
16 e -> true {
17     return true;
18 }
19 e -> false {
20     return false;
21 }
22 e -> if e1 then e2 else e3 {
23     return ((Boolean)e1)?e2:e3;
24 }
```

Most of the specification are similar to Example 2 except for the last component of the evaluator, in which different occurrences of the nonterminal `e` are renamed to avoid ambiguity (Line 22 to Line 24). In general, ESL allows renaming the occurrences of nonterminal on the right hand side of the productions to new names.

The generated EDSL works but may cause a runtime type error at a type cast such as on Line 23 when `e1` is a natural number. To solve this problem, ESL

allows specifying types and typing rules for the EDSL. We add two types, `bool` and `nat`, and typing rules to the specification.

```
syntax
e -> zero
e -> succ e
e -> true
e -> false
e -> if e then e else e

static
type
ty -> bool
ty -> nat
ty -> t : var
typing e : t

----------
e -> zero : nat

e : nat
----------
e -> succ e : nat

----------
e -> true : bool

----------
e -> false : bool

e1 : bool
e2 : t
e3 : t
----------
e -> if e1 then e2 else e3 : t

dynamic
evaluate: Object
e -> zero {
    return 0;
}
e -> succ e {
    return (Integer)evaluate(e)+1;
}
e -> true {
    return true;
}
e -> false {
    return false;
}
```

```
48 e -> if e1 then e2 else e3 {
49     return ((Boolean)e1)?e2:e3;
50 }
```

The types and typing rules are defined in the static section (Line 8 to Line 32). The static section starts with `static` and has two subsections (in this example).

The first subsection (Line 9 to Line 12), which starts with `type`, specifies the grammar of types in the typing rules in a similar fashion as in the syntax section. Here not only do we need to specify the types of the EDSL, which are `bool` and `nat`, but also meta variable `t` used in the typing rules. A meta variable is not part of the type system of the EDSL, but a placeholder for types of the EDSL in the typing rules; in general, all meta variables used in the typing rules need to be defined. The `: var` construct following a production defines a meta variable.

The second subsection (Line 13 to Line 32), which starts with `typing e : t`, consists of typing rules. `e : t` indicate that an EDSL program can have any type. Alternatively, we may specify that an EDSL program must have the `nat` type by `e : nat`. In general, one can write $nt : C$, where $nt$ is a nonterminal defined in the syntax section, and $C$ is a type defined in the `types` subsection.

A typing rule definition consists of zero or more lines of antecedents, a line of dashes, and one line of postcedent. The line of postcedent has the form $P : C$, where $C$ is defined as before and $P$ is a production from the syntax section. Each line of the antecedent has the form $B : C$, where $C$ is defined as before and $B$ is a nonterminal that occurs in $P$. Each typing rule is written in the ESL similar to the way they are usually written as shown in Figure 2.

$$\frac{e : nat}{succ\ e : nat}$$

```
18 e : nat
19 -----------
20 e -> succ e : nat
```

**Fig. 2.** Comparison of a Typing Rule and Its Specification

ECG generates the support code so that only well-typed EDSL program can be compiled. (cf. Section 4.3)

### 3.3 Native Values and Types

The ESL allows using native types in an EDSL specification.

*Example 4.* Native values and types.

```
1 syntax
2 e -> int(i)
3 e -> bool(b)
4 e -> if e then e else e
5
6 static
```

```
7  i = Integer
8  b = Boolean
9  type
10 ty -> t : var
11 typing e : t
12
13 ----------
14 e -> int(i) : Integer
15
16 ----------
17 e -> bool(b) : Boolean
18
19 e1 : Boolean
20 e2 : t
21 e3 : t
22 ----------
23 e -> if e1 then e2 else e3 : t
24
25 dynamic
26 evaluate: Object
27 e -> int(i) {
28     return i;
29 }
30 e -> bool(b) {
31     return b;
32 }
33 e -> if e1 then e2 else e3 {
34     return ((Boolean)e1)?e2:e3;
35 }
```

Recall that each production in the syntax section defines a method. On Line 2 and Line 3, two methods are defined. Unlike in previous examples where the methods do not have any parameter, each one of these two methods has one parameter, whose type is given on Line 7 and Line 8, respectively. A method chain looks like `if().bool(true, BOOLEAN).then().int(1, INTEGER).else().int(0, INTEGER)`, where `BOOLEAN` and `INTEGER` are generated constants used to mark the EDSL type of the subterms. A type marker is required for any subterm whose typing rule has a postcedent in which the type is not a meta variable as shown on Line 13 to Line 17. We discuss how the requirement of these type markers may be eliminated in Section 5.2.

### 3.4 Typing Environments for Higher Order Functions

In this section, we use the simply typed lambda calculus (STLC) with de Bruijn indices as an example to show how to specify a typed EDSL with higher order functions in the ESL.

*Example 5.* STLC terms can be written in a nameless form using de Bruijn indices. For example, the STLC term $\lambda x.x$ can be written as $\lambda 0$, and the STLC

term $\lambda x \lambda y \lambda z.x(yz)$ can be written as $\lambda\lambda\lambda.2(10)$. Furthermore, de Bruijn indices can be represented by Peano numbers, in which the $z$ constructor represents number 0 and the $s$ constructor represents the function $f(x) = x+1$, so that the term $\lambda x \lambda y.yx$ can be written as $\lambda\lambda.z\ sz$. Next, we specify a simple MCS EDSL so that we can write the term $\lambda\lambda.z\ sz$ as a chain of method calls to the following methods: `abs`, `abs`, `app`, `z`, `s`, and `z`.

```
syntax
e -> z
e -> s i
e -> abs e
e -> app e e
i -> z
i -> s i

static
type
ty -> t : var
ty -> t1 : var
ty -> t2 : var
ty -> fun ty ty
environment
env -> E : var
env -> emp
env -> push env ty
typing emp |- e : t
-----------
push E t |- e -> z : t

E |- i : t
-----------
push E t1 |- e -> s i : t

push E t1 |- e : t2
-----------
E |- e -> abs e : fun t1 t2

E |- e1 : fun t1 t
E |- e2 : t1
-----------
E |- e -> app e1 e2 : t

-----------
push E t |- i -> z : t

E |- i : t
-----------
push E t1 |- i -> s i : t

dynamic
```

```
44 ...
```

Let us take a closer look at the static section. A new subsection (Line 15 to Line 18) starting with `environment` is added to specify the grammar for typing environments in the type rules. In this example, a typing environment is modeled as a stack of types. The typing rules include typing environments. A postcedent has the form $A \mid\!- P : C$ and a line of an antecedent has the form $A \mid\!- B : C$, where $A$ is a typing environment and $C$, $B$, and $P$ are defined in the same manner as in Section 3.2.

### 3.5 Parametrized Grammar

The functional nesting style (FNS) is frequently used in functional programming languages. In the FNS, EDSL programs are embedded into host languages as nested functions or constructors. An example of the FNS is `sub(add(cons(1), cons(2)),cons(4))`. The example in Section 1 also uses FNS on Line 2. EriLex supports specifying EDSLs that have both the MCS and the FNS, by utilizing "parametrize grammars".

Let $\mathsf{z}, \mathsf{z}_1, \ldots, \mathsf{z}_n$ denote nonterminals and $\mathsf{a}$ denote terminals.

**Definition 1.** *A* parametrized grammar *is a context-free grammar in the Greibach Normal Form, equipped with an* arity *function that maps every production* $\mathsf{z} \to \mathsf{a}\mathsf{z}_1 \ldots \mathsf{z}_n$ *in the grammar to an integer* $\mathsf{p}$ *between* 0 *and* $n$*;* $\mathsf{z}_1 \ldots \mathsf{z}_\mathsf{p}$ *are parameters (of* $\mathsf{a}$*). A parametrized grammar also requires that nonterminals be divided into two disjoint groups, the parameters and the nonparameters: nonterminals that are not parameters are nonparameters. Terminals appearing in productions of (non)parameters are (non)parameters.*

We usually write $\mathsf{z} \to \mathsf{a}(\mathsf{z}_1 \ldots \mathsf{z}_\mathsf{p}) \ldots \mathsf{z}_n$ if $\mathsf{z}_1 \ldots \mathsf{z}_\mathsf{p}$ are parameters. Disjointness means that a nonterminal can not be both a parameter and a nonparameter.

*Example 6.* For example, $e \to var(i), i \to z, i \to s(i)$ a parametrized grammar, while $e \to var(i), i \to z, i \to s\ i$ is not.

In the generated support code, parameters are translated to formal arguments of methods instead of methods as nonparameters are, as illustrated in the following example.

*Example 7.* An EDSL in the FNS.

```
1  syntax
2  prog -> expr(e)
3  e -> int(n)
4  e -> add(e e)
5  e -> sub(e e)
6
7  static
8  n = Integer
9
10 dynamic
```

```
11 evaluate:Integer
12 prog -> expr(e) {
13     return evaluate(e);
14 }
15 e -> int(n) {
16     return n;
17 }
18 e -> add(e1 e2) {
19     return evaluate(e1) + evaluate(e2);
20 }
21 e -> sub(e1 e2) {
22     return evaluate(e1) + evaluate(e2);
23 }
```

The ESL supports parametrized grammar through a simple form as demonstrated on Line 2 where the nonterminal `e` is made a parameter. We have seen this form in Example 4, where the `int` method has a parameter `i` which has type `Integer`. As shown on Line 4, the ESL also supports more than one parameters. An example of programs in the EDSL is `prog().expr(sub(add(int(1),int(2)), int(4))).run()`.

### 3.6 Name Embedding

EriLex is designed to be able to generate code for multiple host languages. One of the problems of code generation for different languages is that they have different sets of reserved words and naming conventions. For example, `val` is not a keyword in Java, but is one in Scala.

To alleviate this problem, the ESL supports defining "name embeddings" in an EDSL specification which map symbols used in the specification to different symbols in the host language. Name embeddings are defined in a separate optional section at the beginning of a specification. For example, if the EDSL uses `val` and we are generating code for Scala, we can add the following section.

```
1 embedding
2 val=‘val‘
```

Name embeddings can also be used to resolve difference in naming conventions. For example, in Java, the type of integer objects is `Integer`, while in Scala it is `Int`. We can write the following when we are generating code for Scala.

```
1 embedding
2 Integer=Int
```

### 3.7 Interoperability

An EDSL that supports functions can use native methods in the host language as shown in the following example.

*Example 8.* Interoperability.

```
1  syntax
2  e -> cons(n)
3  e -> app e e
4
5  static
6  n : var
7  type
8  ty -> t : var
9  ty -> t1 : var
10 ty -> fun ty ty : fun
11 typing f : t
12
13 n : t
14 -----------
15 e -> cons(n) : t
16
17 e1 : fun t1 t
18 e2 : t1
19 -----------
20 e -> app e1 e2 : t
21
22 dynamic
23 evaluate:Object
24 e -> cons(n) {
25     return n;
26 }
27 e -> app e1 e2 {
28     return ((Method)e1).invoke(null,evaluate(e2));
29 }
```

On Line 6, the `: var` construct marks `n` as nonterminal that can produce anything. On Line 10, the production is marked using `: fun`, which make ECG generate wrapper methods `wrapfun1`, `wrapfun2`, etc. that wrap (unary, binary, etc.) host language methods into EDSL functions. For example, given variable `max` which holds a reference to a reflection object representing the `Maths.max (Double,Double)` method in Java, `wrapfun2(max)` returns an EDSL function of type `func Double func Double Double` so that we can write method chain such as `<Double>app().cons(wrapfun2(max)).cons(0)`.

## 4 EriLex Code Generator

### 4.1 ECG Overview

The ECG takes in an EDSL specification and generates an in memory language independent data structure that represents the support code from which the actual code of the host language is generated. There are two kinds of classes that are generated by EriLex.

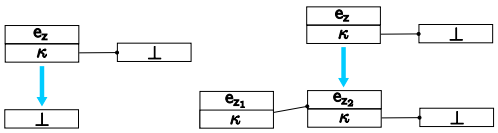
– Utility classes such as `Util`, which provides utility methods and data structures that are largely the same for difference EDSLs.

– EDSL-specific classes consisting of
  - classes that represent syntax and typing rules of the EDSL,
  - classes for ASTs, and
  - evaluators.

The ECG requires that the generation target support basic OO features such as classes and methods and generics to the level of Java. Most mainstream OO programming language would qualify.

In this section, we focus on the classes that represent syntax and typing rules of the EDSL. Other kinds of classes are straightforward to construct.

### 4.2 Generated Support Code for Untyped EDSLs




(a) by $\kappa$ a() (b) by $e_{z_1}\langle e_{z_2}\langle\kappa\rangle\rangle$ b()

**Fig. 3.** Transitions from $e_z\langle\bot\rangle$

The general idea of code generation is that for each grammar of EDSL defined in the ESL, there is a correponding stateless deterministic realtime pushdown automaton (pda for short) that is equivalent to the grammar, and that for that pda, the ESL generates a set of classes representing its transition rules. By transitivity, methods in the generated classes can only be composed in the way that is specified by the grammar.[15]

The pda can be constructed by taking the nonterminals to be the stack symbols and viewing a production $nt \rightarrow t\ nt_1\ \ldots\ nt_n$ as a transition rule that pops $nt$, pushes $nt_1, \ldots, nt_n$, and is labeled $t$, so that each transition rule corresponds to a production. To represent transition rules, the ECG generates for each nonterminal a generic class that has a type parameter and is used to construct types for representing the stacks of pda configurations. The ECG also generates a special class, written $\bot$ (Java name `Bot`), for representing the empty stack. For example, suppose that for nonterminal z, $z_1$, and $z_2$, the generated classes are $e_z\langle\kappa\rangle$,$e_{z_1}\langle\kappa\rangle$, and $e_{z_2}\langle\kappa\rangle$, where $\kappa$ is the type parameter. The type $e_z\langle e_{z_1}\langle e_{z_2}\langle\bot\rangle\rangle\rangle$ represents the pda configuration with z, $z_1$, and $z_2$ on the stack.

Now, onto the representation of the transition rules. In general, a generated method represents a transition rule of the pda. The method name represents the label; the class type in which the method is defined represents the originating configurations; and the return type of the method represents the target configurations. For example, suppose that $e_z\langle\kappa\rangle$ has two methods with signatures shown below

```
1    public κ a();
2    public e_{z_1}<e_{z_2}<κ>> b();
```

Method a represents a transition rule $z \to a$ (that pops z and is labeled a), and method b represents a transition rule $z \to bz_1z_2$ (that pops z, pushes $z_1, z_2$, and is labeled b). Suppose that the originating configuration of the pda is represented by $e_z \langle \bot \rangle$. Calling method a (resp. method b) on an object of this type transits the pda to the configuration $\bot$ (resp. $e_{z_1} \langle e_{z_2} \langle \bot \rangle \rangle$) as shown in Figure 3(a) (resp. 3(b)).

A method chain represents a sequence of transitions in the pda. The type of any prefix of the method chain represents the configuration of the pda as a result of the transitions represented by the prefix. Next, we look at a concrete example.

*Example 9.* We look at a generated class for Example 2. Here we show the method signatures only.

```
1 public class Ee<K> {
2    public K zero();
3    public Ee<K> succ();
4    public Integer run();
5 }
```

It is obvious that `int val = prog().succ().zero().run();` does not generate error messages, while `int val = prog().zero().zero().run();` generates an error message that says that the second `zero` method is not defined.

### 4.3 Generated Support Code for Typed EDSLs

Pdas are not expressive enough for representing both syntax and typing rules. Instead, we utilize pdas with storage[6]. A pda with storage is an extension of a pda that allows attaching "storage" to the stack symbols. For example, in a pda with storage where the stack symbols are exactly the nonterminals, we may attach to them "storage" that are the typing environments and types of the subterms produced by those nonterminals.

The general idea of code generation is that for each set of typing rules (which subsumes the grammar since they are syntax-directed) of an EDSL defined in the ESL, there is a correponding pda with storage that is equivalent to the typing rules, and that for that pda with storage, the ESL generates a set of classes representing its transition rules.[15]

Each typing rule can be viewed as a transition rule of the pda with storage, where the postcedent corresponds to originating configurations while the antecedents correspond to target configurations. The production that is subsumed by the typing rule governs the label and the stack while the types and typing environments govern the storage.

To represent a transition rule, the ECG generates a method. The method name represents the label; the class type in which the method is defined represents the originating configurations; and the return type of the method represents the target configurations.

*Example 10.* Now we look at some of the generated classes for a typed EDSL based on Example 5 extended with a `cons` construct that introduces a native value.

```
1 syntax
2 e -> cons(n)
3
4 static
5 n : var
6 typing
7 E |- n : t
8 ----------
9 E |- e -> cons(n) : t
10 ...
```

Here we show the method signatures only.

```
1 public class fun<t1, t2> {}
2 public class push<t1, t2> {}
3 public class emp {}
4 public class Bot {}
5 public class Util {
6     public static <t> Ee<Bot,t,emp> prog();
7 }
8 public class F<S,T> { ... }
9 public class ID<S> extends F<S,T> { ... }
10 public class Ee<K,t,E> {
11     public t cons(t n);
12     public <E1> Ei<K,t,E> z(F<push<E1,t>,E> cast);
13     public <E1,t1> Ei<K,t,E1> s(F<push<E1,t1>,E> cast);
14     public <t1> Ee<K,t2,push<E,t1>> abs(F<fun<t1,t2>,t> cast);
15     public <t1> Ee<Ee<K,t1,E>,fun<t1,t>,E> app();
16 }
17 public class Ei<K,t,E> { ... }
```

For each terminal symbol that appears in the `type` and `environment` section of the specification, a class is generated as shown on Line 1 to Line 3. But no class is generated for meta variables. The `fun`, `push`, and `emp` classes serve as constructors of host language types that represent EDSL types and typing environments. The `Bot` class represents the empty pda stack. The `F<S,T>` utility class and the `ID<S>` utility class which extends `F<S,S>` are used in some of the generated methods such as `abs` to encode EDSL typing constraints.

The generated class `Ee` has three type parameters. The first type parameter is same as in the support code generated for an untyped EDSL. The other type parameters represent the "storage", where the second type parameter represents the type of (the subterm produced by) the nonterminal and the third parameter represents the typing environment of (the subterm produced by) the nonterminal.

Figure 4 shows the correspondence between the method signature and the typing rule (transition rule) for production `e -> abs e`, where the corresponding

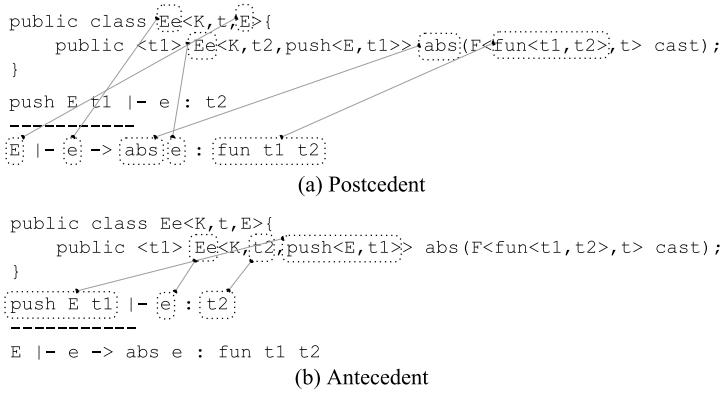



**Fig. 4.** Encoding of Typing Rules

parts of the typing rule and the method signature are connected by connectors. In a pda with storage, the applicability of a transition rule depends on not only the top stack symbol of the originating configuration, but also the "storage" attached to that symbol, which in our application, is the type and typing environment given in the postcedent of the typing rule (recall that the postcedent corresponds to the originating configuration). The encoding of this dependency is a little complex when the type in the postcedent is not a metavariable, as shown in (a) where, when translated to the transition rule, the requirement is that the type in the "storage" attached to the top stack symbol be `fun t1 t2` as specified in the postcedent. Because Java does not support a straightforward way of specifying the structure of a type parameter of the class, the ECG has to generate the `cast` parameter and require the library user to pass in an instance of `ID` when `abs` is called, which says, intuitively, that the type parameter `t` should have the structure `fun<t1,t2>`. [2]

As an example showing how this encoding works, suppose that we have a prefix of method chains `<Integer>prog()` with type `Ee<Bot,Integer,emp>` which represents a stack with only one symbol `e` with attached storage `Integer,emp` , which means that the type of the stack symbol be `Integer`. If we append a method call to `app` to the prefix, the type of the new prefix `<Integer>prog() .<Double>app()` becomes `Ee<Ee<Bot,Integer,emp>,fun<Double,Integer>,emp>`, as shown in Figure 5, which represents a stack with two symbols which are both `e` with attached storage, respectively, `fun<Double,Integer>,emp` and `Integer,emp`, which means that the type of the top stack symbol (resp. second stack symbol) is `fun Double Integer` (resp. `Double`). Therefore, the following

```
1 <Integer>prog().<Double>app().cons(1).run();
```

does not compile and has a type error. In contrast, the following compiles without any type error.

[2] Because of space limit, we do not elaborate the formalized general encoding[15] in this paper.

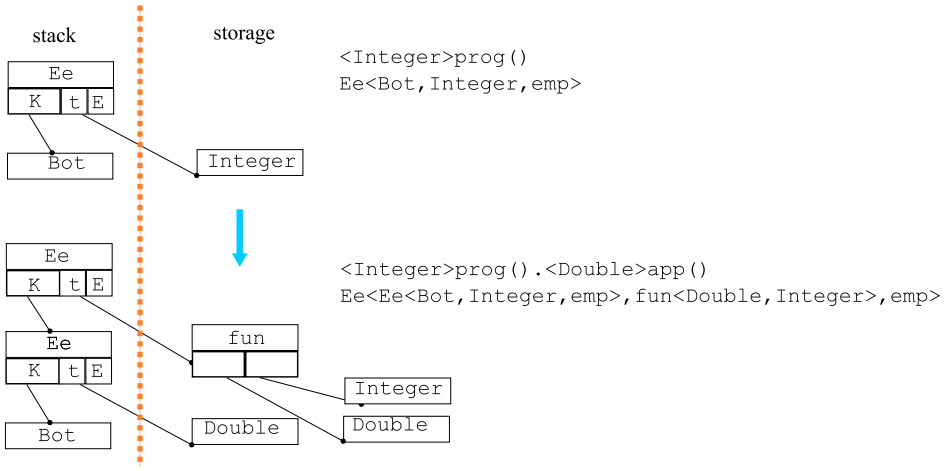



**Fig. 5.** Encoding Pda with Storage

```
ID<fun<Double,Integer>> tyF = new ID<fun<Double,Integer>>();
<Integer>prog().<Double>app().abs(tyF).cons(1).cons(2).run();
```

## 5 Discussions

### 5.1 Tool Reuse

Reuse is one of the fundamental goals of software design. EriLex takes a first step towards editor reuse. Why is editor reuse important? According to the online report from Netbeans Quality Dashboard[2][3], the Netbeans Integrated Development Environment (IDE) has over 4,000,000 line of code. While developer tools such as IDEs are an important factor in the popularity of a programing language, the workload for programming a tool for a new programming language can be prohibitively heavy. Therefore, both the expertise and efforts in the existing tools are simply too large a resource to be left unreused. Three kinds of common tool support – syntax checking during typing, type checking during typing, and auto-completion – are provided in the semantic editor of Netbeans for Java. The MCS effectively establishes the mappings as shown in Table 1, which allows MCS EDSLs to reuse functions provided by the semantic editor without any modification to the IDE.

### 5.2 Improving Usability of EDSLs

It is obvious that the usability of an EDSL is highly dependent on the capabilities of the host language. The previous examples also show some drawbacks of using

[3] Data may change as Netbeans is an open source software under active development.

**Table 1.** Mappings

| Tool Support | auto-completion MCS | | checking during typing | |
|---|---|---|---|---|
| EDSL | parser state | next tokens | syntax | type |
| host language | class type | methods | syntax/type | type |

Java as a host language for EDSLs that cause usability issues for some kinds of EDSL, such as:

1. The symbols "`().`" can not be omitted.
2. Operators can not be use as method names.
3. Without type inference, type arguments such as those for `app` in Example 8, need to be written explicitly.
4. Overloaded methods can not have the same parameter type but different return type. This restricts all grammars to LL(1).
5. Error messages are very difficult to parse or translate to those of EDSLs. If an error occurs, it is very difficult to locate the error.

The usability of the EDSL can be improved with the following improvements of the host language.

1. Make the syntax flexible.
2. Infer type parameter for methods such as the `app` method automatically.
3. Support the `where` construct[10] with which the ECG can generate code without `cast`, for example, `public <t1> Ee<K,t2,push<e,t1>> abs()where fun<t1 ,t2>=t;`
4. Support (nondisjoint) union types so that we can simulate nondeterminism.
5. Provide an interface for writing customized error message generator.

Improvement 1 to 3 are already partially or completely supported by Scala. Improvement 4 has also been shown to be feasible in $C\sharp$[10]. Combining just these improvements would result in much more legible EDSL code. Improvement 5 may be difficult to implement because of the interaction with other features in Java. Improvement 6 can also be implemented in the following manner. We can write a external tool that reads and parses the output of the Java compiler and output the translated error messages. However, because there lacks a "specification" for compiler output, maintaining the tool would not be very easy.

## 6 Related Work

Many methods have been proposed for developing EDSLs, a subclass of DSLs[14,12], in languages such as Haskell[4], MetaOCaml[13], and Scala[9], to name a few. EriLex tries to incorporate some of the techniques and experiences gained from functional programming languages and provides a tool for automatic generation of support code for a range of EDSLs in an OO setting. EriLex carries over some of the techniques for typed representation of EDSL programs[3] developed

for optimization in a functional setting and uses them in a software tool that improves automation and productivity for OO software library design. Compiler compilers[7] generate compilers from formal specifications of programming languages, while EriLex generates support code for EDSLs that reuses tools designed for the host language where extra restrictions are placed. Intentional programming[5,11] generates tools such as editors from language specifications, taking a generative approach, while EriLex takes a reusing approach.

## 7 Summary

EriLex is software tools for generating support code for EDSLs. The ESL has very few constructs yet is expressive enough for a range of EDSLs, which makes EriLex easy to learn and use.

## References


1. jMock. http://www.jmock.org/.
2. Netbeans quality dashboard. http://quality.netbeans.org/sourcelines/summary-teams.html, 2009.
3. Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02*, pages 157–166, New York, NY, USA, 2002. ACM.
4. Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
5. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
6. J Engelfriet and H Vogler. Pushdown machines for the macro tree transducer. *Theor. Comput. Sci.*, 42(3):251–368, 1986.
7. Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, 1999.
8. Max Rydahl Andersen Emmanuel Bernard Gavin King, Christian Bauer and Steve Ebersole. Hibernate reference documentation 3.3.2.ga, 2009.
9. Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *GPCE '08*, pages 137–148, New York, NY, USA, 2008. ACM.
10. Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *OOPSLA '05*, pages 21–40, New York, NY, USA, 2005. ACM.
11. Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors. *DSPG '03, Revised Papers*, volume 3016 of *LNCS*. Springer, 2004.
12. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
13. Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Not.*, 37(9):218–229, 2002.
14. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.
15. Hao Xu. A general framework for method chaining style embedding of domain specific languages. *UNC Technical Report*, 2009.