Eria: Meta Programming with Programmable Inference

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

Eria is a meta language for embedded domain specific languages (EDSLs) which combines object-oriented (OO) programming with logic programming to enable usable embedding of typed object languages in the method chaining style. Most of existing research in EDSLs focuses on improving the efficiency of EDSLs, and the usability of the EDSLs has not been studied much. Eria is designed so that a wide range of EDSLs can be encoded with high usability. The key to high usability embedding of EDSLs is not only type inference but also term inference in the host language. Combining OO features with the Horn Logic, Eria allows users to define rules in the form of Horn clauses on the type level and uses the SLD resolution [19] to instantiate type variables in a type and infer terms that inhabit the type. We also created a decidable restriction of the SLD resolution. In this paper, we introduce the Eria programming language, show several examples of application of Eria, and formally present FE, a core calculus of Eria based on the FGJ [15].

1. Introduction

The concept of embedded domain specific languages (EDSLs) [14] is closely related to software library interface design. A software library usually provides an application programming interface (API) which consists of elements that can be used to write other programs. If the API of a software library is designed so that it mimics a new programming language for a specific domain defined by the functionality of the software library, then this new language is usually refer to as an EDSL.

When discussing EDSLs, we usually use the notion of "object language" and "host language". A host language is a language in which EDSLs are implemented. An object language is a language that is itself unrelated to any host language. An embedding relates an object language to a host language by mapping terms of the object language to terms of the host language. We refer to the image of an object language under an embedding as an embedded language. In this notion, an EDSL is an embedded language.

The traditional ways of embedding are to encode object language terms as host language terms constructed from nested calls to functions or constructors defined according to the object language grammar productions, which we call the functional nesting style (FNS). For example, given an object language, the terms of the object language can be encoded using generalized abstract data types (GADTs) and the dynamic semantics of the object language can be encoded as a function on the GADTs.

The method chaining style (MCS) embedding encodes terms of the object language as method chains of the host language. In an MCS embedding, we view the set of terms of the object language as a language accepted by an automaton and encode this automaton as classes and methods in the host language so that method invocations can be chained together according to a set of rules. The dynamic semantics of the object language can be encoded in the methods.

On one hand, although software libraries sometimes provide MCS APIs in practice, such as in jMock [2] and Hibernate Criteria Query [10], the MCS embedding is studied much less extensively compared the FNS embedding, probably because of lack of term and type inference in object-oriented (OO) programming languages.

On the other hand, most of existing research in EDSLs focuses on improving the efficiency of EDSLs and the usability of the EDSLs has not been studied much. However, usability is important in real world applications. Users who program in a domain specific language (DSL) sometimes understand the application domain better than a general purpose programming language. One of the usages of EDSLs is allowing users not familiar with a host language to program for specific application domains in the host language. When they use an EDSL, they expect similar experiences to a standalone DSL. If the EDSL contains many host language constructs that are unrelated to the application domain, then the EDSL could confuse the users. Therefore, to make an EDSL usable, programs written in the EDSL should look similar to their counterpart written in the standalone DSL (we formalize this concept in Section 2.1).

Built on experiences gained from previous work on formalizing MCS EDSL encoding [38] and implementing EDSL generators [39], Eria takes a new perspective to language design. The main contribution is the following:

- We proposed a definition of usable static language embedding that captures an important aspect of usability of EDSLs.
- Eria combines OO features with logic programming to make it possible to embed typed object languages in Eria in the MCS usably so that the embedded language has a similar appearance to that of the object language.
- Eria allows defining customized error messages.

- Eria allows users to define rules on the type level and uses a decidable variant of the SLD resolution [19] to instantiate type variables in a type and infer terms that inhabit the type.
- We defined a core calculus FE based on FGJ.

This paper is organized as follows. Section 2 introduces preliminaries and discusses motivations of Eria. Section 3 presents features of Eria. Section 4 presents examples of Eria. Section 5 presents FE, a core calculus of Eria. Section 6 discusses related work. Section 7 discusses future work.

2. Preliminaries and Motivation

2.1 Usable Static Language Embedding

The usability of an embedding is a complex problem. We focus on one aspect of usability, which we refer to as static usability, and try to formalize it in this subsection. Intuitively speaking, static usability of an embedding measures the legibility of a program written in the embedded language.

Definition 2.1. A language $\mathcal{L}$ is a 5-tuple

$$(T_{\mathcal{L}}, V_L, Syn_{\mathcal{L}}, Sta_{\mathcal{L}}, Dyn_{\mathcal{L}})$$

$T_{\mathcal{L}}$ is the set of all tokens of $\mathcal{L}$. $V_{\mathcal{L}}$ is the set of all values of $\mathcal{L}$. The set of strings of $\mathcal{L}$, written $strings(\mathcal{L})$, is the set of all finite sequences of elements of $T_{\mathcal{L}}$. $Syn_{\mathcal{L}} \subset strings(\mathcal{L})$ is the syntax of $\mathcal{L}$. $Sta_{\mathcal{L}} \subset Syn_{\mathcal{L}}$ is the static semantics of of $\mathcal{L}$, $Dyn_{\mathcal{L}} : Sta_{\mathcal{L}} \to V_{\mathcal{L}}$ is the dynamic semantics of $\mathcal{L}$. The set of well-formed and well-typed term of $\mathcal{L}$, written $terms(\mathcal{L})$, is just the set $Sta_{\mathcal{L}}$.

For a language $\mathcal{L}$, $Syn_{\mathcal{L}}$ is usually specified by a grammar, and $Sta_{\mathcal{L}}$ is usually specified by typing.

Roughly speaking, the components can be divided into two groups. The first group, a combination of the syntax and the static semantics, defines valid terms of the language, while the second group, the dynamic semantics, defines computation of valid terms. The first group is important in the static usability of a language as it is the "interface" between the language and programmers and the second group is hidden behind the interface of the language.

Definition 2.2. A static language $\mathcal{L}$ is a triple

$$(T_{\mathcal{L}}, Syn_{\mathcal{L}}, Sta_{\mathcal{L}})$$

The components, $strings(\mathcal{L})$, and $terms(\mathcal{L})$ are defined in the same way as in Definition 2.1.

Definition 2.3. Given an static language $\mathcal{O}$ and a static language $\mathcal{H}$, a static embedding $e_{\mathcal{O},\mathcal{H}}$ of $\mathcal{O}$ into $\mathcal{H}$ is an injection from $strings(\mathcal{O})$ to $strings(\mathcal{L})$ such that for every term $t \in terms(\mathcal{O})$, $e_{\mathcal{O},\mathcal{H}}(t) \in terms(\mathcal{H})$ and for every string $t \in strings(\mathcal{O}) \backslash terms(\mathcal{O})$, $e_{\mathcal{O},\mathcal{H}}(t) \in strings(\mathcal{H}) \backslash terms(\mathcal{H})$. We refer to $\mathcal{O}$ as the *object language*, $\mathcal{H}$ the *host language*, and $e_{\mathcal{O},\mathcal{H}}(t)$ the representation of t. We also refer to the embedded object language in the host language $(T_{\mathcal{H}}, e_{\mathcal{O},\mathcal{H}}(Syn_{\mathcal{H}}), e_{\mathcal{O},\mathcal{H}}(Sta_{\mathcal{H}}))$ as the *embedded language*.

A static embedding of a language $\mathcal{O}$ into another language $\mathcal{H}$ maps strings of $\mathcal{O}$ to strings of $\mathcal{H}$ such that a representation of a string t of $\mathcal{O}$ is a term of $\mathcal{H}$ if and only if t is a term.

Terms	e	$\to$	$a \mid x \mid \lambda x.e \mid (e\ e)$
Types	τ	$\to$	$\alpha \mid \tau \to \tau$
Type environments	Γ	$\to$	$\bullet \mid \Gamma, x : \tau$
Values	w	$\to$	$\lambda x.e$

Typing $\Gamma \vdash e : \tau$

$$\Gamma \vdash x : \Gamma(x) \qquad \Gamma \vdash a : \alpha \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Computation $e \longrightarrow e$

$$\frac{e_1 \longrightarrow e_1'}{(e_1\ e_2) \longrightarrow (e_1'\ e_2)} \qquad \frac{e_2 \longrightarrow e_2'}{(e_1\ e_2) \longrightarrow (e_1\ e_2')}$$

$$(\lambda x.e_1\ e_2) \longrightarrow e_1[e_2/x]$$

Figure 1. Simply Typed Lambda Calculus

Definition 2.4. For every function $f : T_{\mathcal{L}_1} \to T_{strings(\mathcal{L}_2)}$, we define a lifting $f^* : T_{strings(\mathcal{L}_1)} \to T_{strings(\mathcal{L}_2)}$ of f to strings: $f^*(a_1 \dots a_n) = f(a_1) \dots f(a_n)$, where $a_i \in T_{\mathcal{L}_1}$ for all $i \in \{1, \dots, n\}$.

A string a is a substring of another string b, written $a \prec b$, if there exist (possibly empty) strings u, v such that $uav = b$.

Definition 2.5. A *usable static language embedding*, or *usable embedding*, is an embedding $e_{\mathcal{O},\mathcal{H}}$ such that there exists a function $f : T_{\mathcal{O}} \to T_{strings(\mathcal{H})}$ such that $f^*(t) \prec e_{\mathcal{O},\mathcal{H}}(t)$ for every $t \in terms(\mathcal{O})$.

A usable static language embedding may map an object language token to a host language string or add a prefix or a suffix to a host language string, but may not reorder, insert, or delete host language strings representing object language tokens.

We all know that when we learn a new natural language, if the new language can be translated word by word, without grammatical changes, to a known language, then it is easier to learn. Similarly, an embedded language is easier to learn, if it can be translated token by token, without grammatical changes, to the object language, given that people know the object language. An embedded language that is easier to learn when people know the object language is more usable. The definition of usable embedding reflects this intuition.

2.2 Simply Typed Lambda Calculus

The most commonly used object language in the literature [3, 4, 28] for discussion of typed EDSLs is a variant of the simply typed lambda calculus (STLC) – the STLC with de Bruijn indices. We will use both as examples of object languages in this paper. In this subsection, we present their definitions used in this paper.

We use the variant of the STLC without type annotations. The STLC with a primitive type α is defined in Figure 1.

The terms include primitive values a, variables x, abstraction $\lambda x.e$, and application $(e\ e)$. The types include a primitive type α and function types $\tau \to \tau$. The typing environments are sequences of pairs $x : \tau$, where the empty sequence is denoted by $\bullet$. We assume that the typing rules are used in type checking, where we have an expected type for a term. The dynamic semantics consists of β reduction rule where the substitution automatically renames bounded variables in e_1 to avoid capture.

One of the variants of the STLC denotes the variables using the de Bruijn notation and Peano numbers. We refer to this variant as the STLC with de Bruijn indices. The terms of the STLC with de Bruijn indices can be defined as

$$e \to i \mid \lambda e \mid (e\ e) \qquad i \to z \mid s\ i$$

where i is a de Bruijn index in which z and s stand for zero and successor, respectively, λe is an abstraction, and $(e\ e)$ is an application. In abstractions, variable names are not explicitly specified; variables are referred to by a Peano number n that refers to the nth closest enclosing lambda binder. For example, the STLC term $\lambda x.x$ is represented by the STLC with de Bruijn indices term λz and the STLC term $\lambda x.\lambda y.(yx)$ is represented by the STLC with de Bruijn indices term $\lambda\lambda(z\ sz)$.

Next, we discuss how to type check STLC terms.

In the STLC with de Bruijn indices, we can model a typing environment as a stack of types. If we denote the sort of all stacks of types by S, the sort of all types by T, then stacks can be represented by an algebra with binary constructor $-,- :: S \times T \to S$ and nullary constructor $\bullet :: S$. Here, we use :: to denote the typing in the algebra and reserve : to denote the object language typing. We also have partial functions $pop :: S \to S$ and $peek :: S \to T$, defined as $pop(x, y) = x$ and $peek(x, y) = y$. There is a correspondence between operators in this algebra and the term constructors of the STLC with de Bruijn indices: λ pushes a type onto the stack, s pops a type from the stack, and z peeks the type on the top of the stack. For example, suppose that we want to type check the term $\lambda\lambda sz$ against type $\alpha \to \alpha \to \alpha$. The first λ binder introduces a type α and push it onto an empty stack, resulting in a stack $\bullet, \alpha$. The second λ binder introduces another type α and push it onto the stack, resulting in a stack $\bullet, \alpha, \alpha$. Then s pop a type from the stack, resulting in a stack $pop(\bullet, \alpha, \alpha) = \bullet, \alpha$. z peeks the type on the top of the stack, given by $peek(\bullet, \alpha) = \alpha$. The whole expression is well-typed.

A typing environment of the STLC with names has to take variable names into account. If we denote the additional sort of all variable names by N, then the typing environment can be represented by an algebra with tertiary constructors $-,-:-:: S \times N \times T \to S$ and nullary constructor $\bullet :: S$. Now, suppose that we want to type check the STLC term $\lambda x.\lambda y.x$ against type $\alpha \to \alpha \to \alpha$. The first λ binder pushes α onto an empty stack, resulting in a stack $\bullet, x : \alpha$. The second λ binder pushes α onto the stack, resulting in a stack $\bullet, x : \alpha, y : \alpha$. For x, we need to compare it with every variable name on the stack, starting from the top of the stack, and find the type that is associated with the first matching variable name. As a result, we do not have the nice correspondence between operators in this algebra and the term constructors here.

2.3 MCS Encoding of Labeled State Transition Systems

To understand the motivation of Eria, we need to understand the idea of MCS encoding of labeled state trasition systems. (Automata can also be viewed as state transition systems.) In this subsection, we briefly discuss how MCS encoding of state transition systems works using examples in Java.

In constract to algebraic data types, state transition systems are coalgebraic structures. In an MCS encoding, we use object types to represent states and methods to represent transitions. For example, the following class definitions in Java model two states A and B, and a transition $m : A \to B$. In the following examples, we are showing only the method signatures.

```
1 class A { B m(); }
2 class B { ... }
```

Given an object `o` of type `A`, invoking the method `m` at `o` returns an object of type `B`, which models transition labeled m from state A to state B.

Given a static language $(T_{\mathcal{L}}, Syn_{\mathcal{L}}, Sta_{\mathcal{L}})$, we view all tokens in $T_{\mathcal{L}}$ as labels of transitions, and construct a state transition system with an initial state and certain final states so that the labels on a finite sequence of transitions for the initial state to one of the final states corresponds to strings in $Sta_{\mathcal{L}}$.

Example 2.6. Suppose that we want to encode a language with integer, addition, and subtraction in the MCS. The corresponding state transition system has two states, $Term$ and Op, and three labels of transitions, val, add, and sub, as shown in the following class definitions.

```
1 class Term { Op val(int i); }
2 class Op { Term add(); Term sub(); }
```

Both the initial state is $Term$ and the final state is Op. The encoding of the term $1 + 1$ looks like

```
1 new Term().val(1).add().val(1)
```

As every method invocation corresponds to a token of the object language, it is obvious that this embedding is a usable embedding.

By generalizing this example, it is obvious that deterministic finite state automata (dfa) can be easily encoded in Java. In fact, a subclass of pushdown automata (pda) called deterministic stateless realtime pushdown automata can be encoded directly in Java using generics.

Example 2.7. Modeling a deterministic stateless realtime pda for the language defined by the following grammar.

$$A \to a\ B\ B \mid z \qquad B \to b$$

The pda has two stack symbols, A and B, and three input symbols, a, b, and z. We define generic classes

```
1 class A<S> { B<B<S>> a(); S z(); }
2 class B<S> { S b(); }
```

The method `a` represents transitions that consumes a, pops A from the top of the stack, and pushes two B's onto the stack. The method `z` represents transitions that consume z and pops A from the top of the stack. The method `b` models transitions that consume b and pops B from the top of the stack. In this embedding, every method invocation corresponds to a token of the object language. Hence this embedding is a usable embedding.

Typed languages put more restrictions on what valid terms in the language are – not only do they need to conform to the grammar, but also be well-typed. The corresponding state transition systems need to include not only the parser states, but also the states of the type checker.

In particular, we can model some typed languages using pdas with storage. The state of such an automaton is a combination of a stack (the parser state) and storage (the state of the type checker). The following example shows how to encode a typed language in Java.

Example 2.8. We encode a very simple typed language consisting of two type, string and integer, two binary operators, `eq` and `neq` and the typing requirement that the operands of an operator have the same type. In a pda with storage for this language, the "storage" stores the type of the first operand. In the following class definitions, the storage is represented by the type parameter `T`.

```
class A1<T> { B<T> val(T x); }
class B<T> { A2<T> eq(); A2<T> neq(); }
class A2<T> { Bot val(T x); }
class Bot { ... }
```

The method `val` of `A1` introduces the first operand, the type of the operand goes into the type parameter `T`, which is then carried over through the type parameters of `B` and `A2`, and unified with the type of the second operand. For example, the following compiles without errors.

```
new A1<Integer>().val(1).eq().val(1);
```

while the following does not

```
new A1<Integer>().val(1).eq().val("1");
```

This embedding is still a usable embedding, but in general, MCS embedding in Java may not be a usable embedding. We will see an example of this in the next subsection.

It is obvious that most languages have corresponding state transition system that models the languages. When encoding such a state transition system, we need to ask the following questions:

1. What are the states?
2. What are the transitions?
   (a) What are the source states and target states of a transition?
   (b) What are the rules for when this transition is applicable?

Answers to these questions determine types and typing constraints needed in an encoding of the state transition system. For example, for dfas, the states are finite, and have no further structure, and so do the transitions; the applicability of transitions are solely based on the source state. Therefore, classes without any type parameter and methods without parameter are enough. For pdas, however, the states have stack structures, and the transitions may modify the stack. Therefore, more complex types are needed to encode pdas.

2.4 Comparison between GADT and MCS Encoding

Let's start from the GADT encoding of the STLC with de Bruijn indices. We encode the STLC with de Bruijn indices terms as GADT terms in Java. We can write the following interfaces and classes.

```
interface push<E,t> { }
interface emp { }
interface fun<S,T> { }
interface Term<E,t> { }
class Z<E,t> extends Term<push<E,t>,t> {
  Z();
}
class S<E,t,t1> extends Term<push<E,t1>,t> {
  S(Z<E,t> i);
}
class Abs<E,t1,t2> extends Term<E,fun<t1,t2>> {
  Abs(Term<push<E,t1>,t2> e);
}
class App<E,t1,t2> extends Term<E,t2> {
  App(Term<E,fun<t1,t2>> f,Term<E,t1> e);
}
```

These classes encode a GADT in which only representations of (well-typed) STLC with de Bruijn indices terms can be constructed.[1]

Suppose that we want to encode the term $\lambda\lambda(z\ sz)$. Let us assume for now that it is possible to improve the type inference of Java so that the type parameters are inferred, and see if we can get a usable embedding with powerful type inference. The encoding looks like

```
Abs(Abs(App(Z(),S(Z()))))
```

But this is not a usable embedding of $\lambda\lambda(z\ sz)$, because of the the representation uses host language delimiters such as parentheses and commas to delimit constructs of the embedded language. A usable embedding would map the STLC term to something like the following.

```
Abs Abs App Z S Z End
```

The trailing `End` is not a problem. However, this code is inherently ambiguous without further language level rules such as those in function programming languages because the parser does not know the arity of the constructors, which is part of the types of the constructors.

MCS embeddings do not have this problem, as in an MCS embedding, the host language parser is used as the embedded language lexer and the host language type checker is use to parse the embedded language instead.

In the MCS, we use a combination of ideas of Example 2.6 and Example 2.8.

First, let us recall the syntax and typing of the STLC with de Bruijn indices from Section 2.2. In a state transition system for this language, a state is a pair consisting of a stack and a (current) position in a term of the EDSL (a host language method chain). In the stack, every stack symbol has its own storage that stores the typing environment and type of some subterm. The storage of the top stack symbol stores the typing environment and type of the subterm at the current position. This state transition system is a variant of pdas with storage.

There are two kinds of stack symbols in this automaton, represented by class `Term` and class `EOA`, respectively. When they are on the top of the stack, `Term` represents states where the expected token in the current position is a λ, an opening bracket, a constant, or a variable. The second one is states where the expected token in the current position is a closing bracket.

[1] The functional representation of this GADT is commonly used in the literature.

The type parameter `K` represents the next symbol on the stack. The type parameters `t` and `E` represent the storage attached to the corresponding stack symbol, where `t` represents the type of the subterm at the current position, and `E` represents the typing environment of that subterm.

```
interface push<E,t> { }
interface emp { }
interface fun<S,T> { }
interface eq<S,T> { }
class re<T> impliments eq<T,T> { }
class Term<K,E,t> {
  K z(eq<E,push<E1,t>> p);
  <E1,t1> Term<K,E1,t> s(eq<E,push<E1,t1>> p);
  <t1,t2> Term<K,push<E,t1>,t2> abs(eq<t,fun<t1,t2
      >> p);
  <t1> Term<Term<EOA<K>,E,t1>,fun<t1,t>> app();
}
class EOA<K> { K end(); }
```

Again, we assume that we have type inference that can infer all type parameters. The encoding could be written as

```
new Term().abs(new re()).abs(new re()).app().z(new
    re()).s(new re()).z(new re()).end()
```

We still have a problem: the method arguments.

Why do we need these method arguments? As the names of their types suggest, these method arguments are proofs that two types are equal. For example, the method `abs` requires that the type `t` be a function type, otherwise an abstraction would not be well-typed. Why can not we express this property as in the GADT encoding, where the `Abs` constructor did not require any argument like this? The answer lies in the type of the `abs` function. The type can be written as

```
Term<K,E,t>→eq<t,fun<t1,t2>>→Term<K,push<E,t1>,
    t2>
```

while what we want is

```
Term<K,E,fun<t1,t2>>→Term<K,push<E,t1>,t2>
```

However, this can not be achieved in Java because the first parameter of a method is always implicitly bound to `this` which has type `Term<K,E,t>` where the type parameters `K`, `E`, and `t` are universally quantified.

To summarize, even if we have strong type inference, the GADT encoding still needs the host language delimiters because it utilizes the host language parser to parse the embedded language and host language parsers do not know type information; the MCS encoding utilizes the host language type checker to parse the embedded language, but needs the method arguments for proofs of type equality. The MCS encoding example is a usable embedding according to Definition 2.5, only if we have perfect type inference that infers all type arguments (which we do not in Java, Scala, or C#) and the method arguments are all `new re()` which is not the case for more complex EDSL such as those examples in Section 4.

2.5 The Challenges

Type Markers By "type markers", we refer to host language constructs such as method arguments or type arguments that are not part of an object language but are required in host language terms representing terms of the object language for the purpose of type checking those terms only, such as those in the previous subsection. As discussed in the previous subsection, the main obstacle to usable embedding of typed object languages in the MCS is type markers. However, it is very difficult to get rid of them in Java or similar OO programming languages.

Nameless Representation Although the STLC with de Bruijn indices has its theoretical significance, it is very difficult to write large programs in it, especially when a function has a long list of parameters. In contrast, in real world programming languages, parameters can be given meaningful names to help programmers remember their roles in the program. Usable embedding of the STLC with named variables is a step forward towards usable EDSLs.

The main difficulty of encoding the STLC with named variables is that in the STLC with named variables, we do not have the nice correspondence as in the STLC with de Bruijn indices, as discussed in Section 2.2. Automatic generation of proofs of type equality is not enough. Therefore, the host language needs to be "smart" enough to encode the rules for looking up variable types in typing environments. For generality, these rules should not be built into the host language, but the host language should allow defining user defined rules. It turns out that, in many OO programming languages, usable encoding of these rules is very difficult.

Complexity Encapsulation Apparently, if we can design a language whose type system can model more complex and more powerful state transition systems, then it is possible to encode a wider range of EDSLs in the language. When combined with the usability constraint, this means that the type and term inference algorithm of the host language needs to be power enough to encapsulate the complexity and hide them from the EDSL users. We do not want to require EDSL users to write explicit proofs in their EDSL programs.

3. Eria

3.1 Design Guidelines

We would like to design a nonfunctional[2] OO programming language in which we can create a general framework that allows constructing usable static language embedding of a wide range of object languages.

Given the challenges, we set the following high-level design guidelines for the current version of Eria:

1. Flexibility: we want to provide EDSL designers with the flexibility of customizing the type inference of Eria by enabling EDSL designers to defined rules on the type level to express EDSL syntax and typing rules.
2. Usability: EDSL programs should be easy to read and "type markers" free, so that the EDSLs can be usably embedded.
3. Generality: Eria should be a general purpose programming language. The new features should be useful in both EDSLs and programming in general.

The following is an example of what the encoding of the STLC looks like in Eria.

Example 3.1. In Eria, we can construct a usable static language embedding of the STLC with named variables such that terms such as $(\lambda x.x\ 2)$ can be encoded as Eria terms that look like `[ λ#x:#x 2 ]`, while Eria strings representing

[2] In fact, we do not need the host language to directly support higher-order functions.

STLC strings such as $(\lambda x.x\ \lambda x.x)$ (which is not typable in the STLC) or $(\lambda x.y\ 2)$ (which is open) are not typable.

3.2 Overview

Eria is very similar to Java or Scala. Eria supports types defined as interfaces and classes in a similar way as Java. Every value in Eria is an object, which is similar to Scala. Basic types, such as integers, floating point numbers, and booleans, are supported as predefined classes. There is also a `null` value that inhabits every type. Eria supports most expressions and statements that are supported in Java. The syntax and semantics of statements, such as the `if` statement, the `while` statement, and the `return` statement, are very similar to Java and are not elaborated in this paper. Eria has several important extensions. In this section, we informally introduce features of Eria, focusing on those that are not present in Java.

The code examples follow the following conventions unless otherwise specified: `D`, `E`, `F` are class or interface names; `S`, `T` are type variables; `m` is a method name; `f` is a field name; `Int` is the type of integer objects.

3.3 Interfaces and Classes

Let us first look at interfaces and classes. Apart from the syntactical differences, interface definitions in Eria are essentially the same as those in Java.

In an interface, a method declaration starts with the `def` keyword, followed by a method signature and an optional where clause. We postpone the discussion of the where clause to Section 3.7.2.

The method signature starts with an optional type parameter list, followed by a method name, a parameter list, a colon, and a return type. For example,

```
def {T} m{x:E}:F
```

The equivalent Java method declaration is

```
<T> F m(E x);
```

An interface definition may contain method declarations. For example, the following Eria code defined an interface with a method.

```
interface D {
  def {T} m{x:E}:F
}
```

The equivalent Java definition is

```
interface D {
    <T> F m(E x);
}
```

Class definitions in Eria are also very similar to those in Java. A class definition may contain field definitions, constructor definitions, or method definitions. A field definition starts with the `var` keyword. For example,

```
var f:T
```

The equivalent Java definition is

```
T f;
```

A constructor definition starts with the `def` keyword. For example,

```
def D{x:T} { f = x; }
```

The equivalent Java definition is

```
D(T x) { f = x; }
```

A method definition also starts with the `def` keyword. For example,

```
def {S} m{x:S}:S { return x; }
```

The equivalent Java definition is

```
<S> S m(S x) { return x; }
```

To put them together, for example, the following Eria code defines a class with a field, a constructor, and a method.

```
class D<T> {
  var f:T
  def D{x:T} { f = x; }
  def {S} m{x:S}:S { return x; }
}
```

The equivalent Java definition is

```
class D<T> {
  T f;
  D(T x) { n = x; }
  <S> S m(S x) { return x; }
}
```

Eria has a special kind of methods called the E methods. The E methods are declared in an interface without a method name and only one parameter, parenthesized. For example,

```
interface D {
  def (x:Int):Int
}
```

The E methods are defined in a class without a method name and only one parameter, too. For example,

```
class D {
  def (x:Int):Int { return x * x; }
}
```

Unlike a regular method which is invoked by the method name, an E method can not be invoked by a method name. (more on this later)

Eria also allows using operator strings as method names, following the Scala convention. For example,

```
class D {
  def &&{x:Int}:Int { return x * x; }
}
```

3.4 Object Creation

Constructors are invoked in Eria in a way similar to in Java, except that arguments of constructors are surrounded by braces. For example, suppose that we have class definition

```
class D<T>{
  def m{x:T}:T { ... }
}
```

We can create an object of this class using the following expression

```
new D<Int>{}
```

The type argument of the type constructor can be omitted. For example,

```
new D{}
```

The equivalent Java expression is

```
new D<Int>()
```

Eria also support Java-style anonymous inner class. For example,

```
new D<Int>{} { def m{x:Int}:Int { ... } }
```

The types in a method of the inner class can be omitted if the method overrides a method that can be uniquely determined by its method name in the super class. For example,

```
new D{} { def m{x} { ... } }
```

The equivalent Java expression is

```
new D<Int>() { Int m(Int x) { ... } }
```

3.5 Method Invocations

Methods are invoked in Eria in a way similar to in Java, except that type arguments and arguments of methods are surrounded by braces. For example,

```
e.{Int}m{1}
```

The equivalent Java expression is

```
e.<Int>m(1)
```

The type parameters may be omitted in method invocations. For example,

```
e.m{1}
```

Unlike a regular method which is invoked by the method name, an E method is invoked by just writing the method argument (without needing the parenthesis). For example, suppose that we have class definition

```
class D {
  def (x:Int):Int { return x * x; }
}
```

Given a variable `a` of type `D`, to invoke the E method defined in the class, we just write `a.1`.

If the parameter is a variable or a complex expression, then the expression should be enclosed in parentheses. For example, in order to invoke the E method on `1+1`, we should write `a.(1+1)`, but not `a.1+1`.

3.5.1 Dotless Form

In the discussion so far, method invocations are written in full, which we call the "regular form". For example, `e.{Int}m{1}.n{}`. In Java, the equivalent method chain looks like the following: `e.<Int>m(1).n()`. To provide more flexibility, Eria allow another form of method invocation, called the "dotless form". In the "dotless form", the same method chain is written as `e m{1} n`, where the type parameter, the dots, and the empty braces of method invocations are omitted. In the "regular form", the dots play a central role in a method chain as delimiters of method invocations. Without the dots the association of the arguments may be ambiguous. For example, in `e m {X} n`, `{X}` may be either a type argument of `n` or an argument of `m`. To eliminate ambiguity, Eria allows only arguments, but not type arguments in the dotless form. [3]

The "dotless form" and the "regular form" can be mixed in one method chain. For example, `e.{Int}m{1} n`.

[3] Because Eria does not allow directly accessing fields from outside an object, there is no ambiguity between methods with on parameters and fields.

In comparison, Scala also allows omitting parentheses of method invocations that have some arguments, where ambiguity is resolved with additional rules, as illustrated by the following example: without the additional rules, the expression `e m n` can be parsed as both `e.m().n()` and `e.m(n)`; but in Scala it is parsed as the latter. To let Scala parse the expression as the former, we need to write `e m() n`. One of the purposes of the Scala style parsing is allow defining binary operators as methods.

The Eria way of parsing makes long method chain more succinct. For example, `a b c d e` means `a.b(c).d(e)` in Scala but `a.b{}.c{}.d{}.e{}` in Eria. The equivalent to `a.b().c().d().e()` in Scala is `a b() c() d() e()`.

Eria also supports defining binary operators as methods, by using E methods. For example, we can define

```
class Int {
  def +{}:Int2 { return new Int2(this); }
}
class Int2 {
  var a:Int
  def Int2{x:Int} { a = x; }
  def (Int x):Int { return x * x; }
}
```

Given the definitions, we can write `1 + 2`, which is parsed as `1.+{}.2` where the second method invocation invokes the E method.

3.5.2 Discussion

The use of braces, parentheses, and angled brackets clarifies the roles of the three sets of symbols.

Braces in Eria expressions represent modifications. Each method invocation in Eria can be though of as a endocentric construction in linguistic terms, with the method name being the head, and the expression or types between the braces being the modifiers. In particular, type arguments are pre-modifiers, while arguments are post modifiers. In a chain of methods, the dots delimits endocentric constructions and a type or an expressions between braces modifies the method name next to it.

Using braces to delimit modifiers frees parentheses from being both representing modification and precedence. Parentheses in Eria expression represents only precedence. Expressions between the parentheses has higher precedence then those outside the parentheses.

Similarly, angle brackets are used only after type constructors. They are a coherent part of a type. In contrast, braces are used in method invocations. They delimits type parameters which are modifiers of a method name.

3.6 Labels

Labels are a special type of values in Eria. (Not to be confused with labels in a labeled state transition system. Unless specified otherwise, this is default meaning of "label" in the following discussion.) Eria supports two label constructors: creating an empty label, written ε, and appending a label character l to a label L, written $L\#\#l$. Eria allows using digits, letters, and underscores in labels. These characters are called label characters. Because the basic syntax for creating long label are quite cumbersome, Eria provides syntax sugar for long labels and appending multiple characters to a label: $\#l_1 \ldots l_n$ and $L\#\#l_1 \ldots l_n$. For example, `#abc123` and `#abc##123`.

In Eria, labels have their own small type system. The type of a label depends on its value. Each label $\#l_1 \ldots l_n$ has type

`#`$l_1 \ldots l_n$ which is itself. This makes every label type in Eria a singleton type. Label types can contain variables, written `?`X , which may occur only in the prefix position. For example, `?`X`##`l. Label type variables can only be substituted by label types. Label types, however, can be part of a larger type that are not label types. For example, C`<#`$l_1 \ldots l_n$`>` where `C` is a normal type constructor.

Labels can be passed around as method arguments. For example, we can define a class representing a variable whose constructor takes in a label.

```
class Var<?L> {
  var ?L:label
  def Var{l:?L} { label = l; }
}
```

The expression `new Var{#x}` has type `Var<#x>`. The expression `new Var{#y}` has type `Var<#y>`. They have different types.

Appending can be used to generate fresh labels, as shown in the following method definition.

```
def {?L} gen{l:?L}:?L##1 { return l##1; }
```

3.7 Programmable Inference

OO programming languages such as Java and Scala support customized inference on the type level to some extent: programmers define the subtyping relation when defining classes and interfaces. Programmable inference in Eria complements it with a more powerful form of customized inference based on the Horn Logic.

3.7.1 Implicit functions

Eria allows users to defined implicit functions, which are defined outside any class and similar to static methods in Java. The implicit functions are defined using the `implicit` keyword. Otherwise the syntax is just like method definitions, as demonstrated in the following examples.

Types of implicit functions are analogous to facts and rules in a Prolog [31] program. If an implicit function does not have parameters, then it corresponds to a fact. The type t of the implicit function corresponds to the literal in the fact, for example, the following

```
implicit z{}:nat<zero> { ... }
```

can be used to represent the fact that `zero` is a `nat`.

If an implicit function has one or more parameters, then it corresponds to a rule. The codomain of the function type corresponds to the head of the rule, and the domain of the function type corresponds to the goals of the rule. For example, the following

```
implicit {N} s{p:nat<N>}:nat<succ<N>> { ... }
```

can be used to represent the rule that `succ<N>` is a `nat` if `N` is a `nat`. There can be multiple parameters, which corresponds to multiple goals.

These types clearly correspond to the following Prolog program.

```
nat(zero).
nat(succ(N)) :- nat(N).
```

In this program, the first line is a fact. The second line is a rule, with `nat(succ(N))` being the rule head, and `nat(N)` being the rule body.

3.7.2 Implicit Parameters

Methods may take in implicit parameters. Implicit parameters are defined in a `where` clause at the end of a method signature. An implicit parameter may have a function type. Such an implicit parameter can used as a function. However, higher-order function types are not supported. As syntax sugar, if the function type has arity one, then the parameter types need not to be enclosed by braces. For example,

```
class D {
  def m0{x:String}:String where
    p:{}→String { return p{}+x; }
  def m1{x:String}:String where
    p:String→String { return p{x}; }
  def m2{x:String,y:String}:String where
    p:{String,String}→String { return p{x,y}; }
}
```

Unlike a regular parameter which is provided by the users when a method is invoked, an implicit parameter is not provided by the users but inferred by the compiler by viewing its type as a Prolog query and its value as a proof. Implicit functions are used to construct inferred arguments for implicit parameters. For example, given the implicit function definitions from Section 3.7.1 and a class definition

```
class D {
  def m{} where p:{}→nat<zero> { ... }
}
```

where we defined an implicit parameter `p` of type `{}→nat<zero>`, the value of the implicit parameter inferred from its type is `z`.

For another example,

```
class D {
  def {N} m{} where
    p:nat<N>→nat<succ<succ<N>> { ... }
}
```

The inferred argument is a function that takes in some argument x and returns `s(s(`x`))`.

There is a special binary predicate, the equality predicate $\equiv$, that is true if two types are unifiable with each other. For example,

```
class D {
  def {S,T} m{} where p:S≡T { ... }
}
```

In this example, the argument `p`, if found, will always be an identity function and will have type `S→T`.

In Prolog, there can be more then one goals in a query. Analogously, there can be more than one implicit parameters in a method definition in Eria. The implicit parameter declarations can be connected using either a comma or a semicolon.

A comma allows backtracking to the implicit parameter that precedes the comma if no solution is found for the implicit parameter that follows the comma. For example,

```
class D {
  def {N} m{} where
    p:nat<N>,q:succ<zero>≡N { ... }
}
```

The first solution for `p` makes `N=zero`. However, this falsifies `succ<zero>≡N`. Eria backtracks and makes `N=succ<zero>`, which satisfies `succ<zero>≡N`.

A semicolon does not allow backtracking, which corresponds to a cut in Prolog. For example,

```
class D {
  def {N} m{} where
    p:nat<N>;q:succ<zero>≡N { ... }
}
```

The first solution for p makes N=zero, which falsifies succ<zero>≡N. Eria does not backtrack and type checking fails.

The algorithm used in Eria to infer arguments for implicit parameters is based on the SLD resolution [19], the same algorithm behind Prolog. In Eria, the inferred argument, if any, is always the first solution found by the SLD resolution, even though there may be more than one solutions to an implicit parameter. Eria always infers arguments for implicit parameters locally when it type checks a method chain, which means that there is no backtracking between two method calls, or equivalently, that there is always a semicolon at the end of each where clause.

Implicit parameters are useful in various occasions, such as user defined coercion.

Example 3.2. Automatic dereferencing.

```
implicit {T} deref{r:Ref<T>}:T { ... }
class Number {
    def {T} get{l:T}:Number where p:T→Number {
        return p{l};
    }
}
```

The number of dereferences is not limited to one. For example, we can write new Number{}.get{new Ref{new Ref{new Ref{1}}}}. Then p has type Ref<Ref<Ref<Number>>>→Number, whose value may be inferred to be a function that takes in a parameter x and returns deref{deref{deref{x}}}. One of inconveniences is that subtyping is not considered in the inference, in order to obtain decidable inference. However, we can easily workaround this by adding subtyping as implicit functions. For example, if we defines

```
class RefImpl<T> extends Ref<T> { ... }
```

then we can define

```
implicit {T} up{r:RefImpl<T>}:Ref<T> { return r; }
```

The presence of semicolons generally means that we could fail even if a solution can be found. If we were to solve a set of constraints, then this is not a desirable side effect. However, this feature has some important usage scenario in EDSL design, as demonstrated by the following example.

Example 3.3. Suppose that we model an object language typing environment as an algebra with the following sorts: stacks S, types T, and names N, and constructors: $push : S \times N \times T \to S$ and $emp : S$. We allow repeated names in a stack and the type of the topmost matching name is the current type of the name. We can encode lookup as specified by the following rules of the object language using implicit parameters.

$$\frac{lookup(E, N) = t}{E \vdash var(N) : t} \quad (\textsc{Var})$$

where $lookup$ is defined by

$$\frac{E = push(E1, N, t)}{lookup(E, N) = t}$$

$$\frac{E = push(E1, N1, t1) \qquad N1 \neq N}{lookup(E, N) = lookup(E1, N)}$$

First, let's translate the rule (VAR) to a state transition by answering Question 2 in Section 2.3. Given a stream of tokens $t_1\ t_2\ \ldots\ t_n\ \ldots$ representing an object language program, each transition consumes one of the tokens in the state transition system. Suppose that the tokens $t_1, \ldots, t_{n-1}$ are consumed and the next token is t_n and a state is a stack of nonterminals, with "storage". (a) On the top of the originating state's stack, there should be the nonterminal that produces the subterm s that starts at t_n with its storage being the expected type and the typing environment of that subterm; the targeting state's stack should has nonterminals for the subterms of s on the top with their storage being types and typing environments for those subterms. (b) A transition $var(N)$ can only be applied if the name N has the expected type, according to the typing environment.

In a first attempt, we may be tempted to encode them as the following, where the three type parameter of class Ee, K, t , and E, model, as described above, the stack of nonterminals, the expected type, and the typing environment, respectively.

```
implicit {t1,E,N} pop{e:push<E,N,t1>}:E { ... }
class Ee<K,t,E> {
  def {?L,X} var{l:?L}:K where
    p:E→push<E1,?L,t> { ... }
}
```

However, this encoding is incorrect. Suppose that we have an expression o with type Ee<...,D1,push<push<emp,#X,D1>,#X,D2>. When translated in to the object language, this type means that the current type of #X is D2, and we are expecting an expression of type D1. Therefore, o.var{#X} should not be typable. However, o.var{#X} generates no type error. To see why, because E→push<E1,?L,t> is not directly solvable with E=push<push<emp,#X,D1>,#X,D2>, ?L=#X, and t=D1, the implicit function pop is automatically applied, generating type push<emp,#X,D1>, which makes E→push<E1,?L,t> solvable. Intuitively, we are looking up by both name and type, not just by name. Therefore, the encoding is incorrect. A semicolon comes in handy here.

```
class Ee<K,t,E> {
  def {?L,X,t1} var{l:?L}:K where
    p:E→push<E1,?L,t1>;q:t1≡t { ... }
}
```

Now, we are just looking up by name. Indeed, o.var{#X} generates a type error, because solving E→push<E1,?L,X> generates the following solution t1=D2, which will fail the second constraint t1≡t because t1=D1 but t=D2.

There is another subtlety that needs more explanation here.

Example 3.4. Suppose that we want to write a method that adds a variable to a larger EDSL term. The following

```
  def {K,t,E} append{s:Ee<K,t,E>}:K {
    return Ee var{#x};
  }
```

is not typable, because we assume that this method is applicable for all `K`, `t`, and `E`, which is apparently false. The typable method is the following

```
def {K,t,E,v,G} append{s:Ee<K,t,E>}:K where
  p:Ee<K,t,E>→Ee<K,t,push<G,#x,v>>;
  q:Ee<K,t,push<G,#x,v>>≡Ee<K,t,push<G,#x,t>> {
  return q{p{s}} var{#x};
}
```

Here, we use the where clause to lift the requirements of the `var` method to requirements of the `append` method.

3.7.3 Order

There is another issue: decidability of type checking. Without any restriction, SLD resolution on Horn clauses, which is used for inferring implicit arguments, is semidecidable. Eria allows a simple restriction based on well-founded order to make implicit argument inference decidable, which we describe in this section.

We require that for any implicit function of type $(S_1, \ldots, S_n) \to T$, the number of type constructors and variables in T is less than or equal to that in S_k for some k. For example, in Example 3.2, the function `deref` has type `Ref <T>→T`, where `Ref<T>` has two symbols in it and `T` has one symbol in it.

For types of same size, Eria tries to construct a dictionary order of types, such that every non nullary implicit function reduces some type to a smaller one. If such an order can not be constructed, then type checking fails.

To add more flexibility, Eria allows some type parameters of a type constructor to be counted zero or more than one times when counting the number of symbols. This is useful when we want to defined functions that model upcast such as `upcast{a:String}:Comparable<String>`. The number of symbols in `String` is always less than the number of symbols in `Comparable<String>`, unless we count the type parameter of `Comparable` zero times. We can specify this in Eria by

```
order Comparable<T> {}
```

As another example, assuming that we want to count the parameter twice when counting the number of symbols, we could write the following.

```
order Comparable<T> {T,T}
```

We postpone formal discussion of how this restriction leads to decidability to Section 5.2.3.

3.7.4 Interaction with other OO Features

The interaction between implicit parameters and subclassing is similar to method parameters – when a method with implicit parameters is overridden, the overriding method should have the same list of implicit parameters with the same types.

In the current version, implicit parameters are used to restrict type arguments, but not to determine which method to dispatch. Inference happens only after all normal type checking are finished and a unique candidate is determined for a method invocation.

3.8 Customizable Error Messages

When using a general purpose programming language as the host language for an EDSL, error messages produced by the host language compiler are usually less relevant to the EDSL, which usually results in error messages that do not match errors in EDSL terms. Customizable error messages bridge the gap between the host language and EDSLs.

Customized error messages are defined using the `error` keyword or the `onerror` keyword. We demonstrated their usage using the following examples. The `error` keyword is used to defined error messages for invoking methods that are undefined. For example,

```
class D {
  def _ error ("message "+_)
}
```

In this form of error messages, the variable `_` is bound to the method name that is undefined. The variable can be concatenated with strings to form error messages. The `onerror` keyword is used to define error messages for parameters or implicit parameters. For example,

```
class A {
  def {T} var{l:T onerror ("message "+T)}:T where
    S→T onerror ("message "+T) { ... }
}
```

The type variables can be concatenated with strings to form error messages.

Next, we look at some examples in EDSLs. The following code defines error messages for unexpected tokens, undefined variables, and type mismatch of variables in an EDSL.

```
class Ee<K,t,E> {
  def _ error ("unexpected token "+_)
  def {E1,?L}var{l:?L}:K where
    E→push<E1,?L,t1> onerror
      ("undefined variable "+?L);
    t1≡t onerror
      ("type mismatch for variable "+?L) { ... }
}
```

4. Examples

4.1 Well-typed Code Generator

Example 4.1. A well-typed code generator guarantees that the generated code is well-typed statically, which is useful when, for example, the code is generated on the fly on a server. We look at a well-typed code generator for the STLC.

First, we define Eria types representing STLC typing environments, where the typing environments are modeled as a stack of pairs of labels and types.

```
interface push<E,?L,t> {
  def pop{}:E
}
interface emp { }
```

Next, we define an Eria type representing function types of the EDSL.

```
interface fun<t1,t2> { }
```

We use the following utility class to build the generated code.

```
class Builder {
  var prefix:String
  def Builder{} { prefix = ""; }
  def append{s:String}:void { prefix = prefix+(s);
      }
  def toString{}:String { return prefix; }
}
```

Next, we define classes that represent an automaton that accepts terms of the EDSL, similar to Section 2.4. A state of this automaton is also a pair consisting of a stack and a (current) position in a term of the EDSL (a host language method chain). In the stack, every stack symbol also has its own storage that stores the typing environment and type of some subterm, with the storage of the top stack symbol stores the typing environment and type of the subterm at the current position. In Section 2.4, the automaton had only two stack symbols. For the STLC, we have four kinds of stack symbols in this automaton, represented by class `Ee`, class `Er`, class `Ev`, and class `Ed`, respectively. When the first symbol is on the top of the stack, the expected token in the current position is a λ, an opening bracket, a constant, or a variable (except those following a λ). For the second one, the expect token is a closing bracket. For the third one, the expected token is a variable following a λ. For the fourth one, the expected token is a colon. The type parameter `K` represents the next symbol on the stack. The type parameters `t` and `E` represent the storage of the stack symbol, where `t` represents the type of the subterm at the current position, and `E` represents the typing environment of that subterm. The applicability of transitions are more complex than a pda, because now we need to lookup a name from the typing environment, which means that we need to infer a term that may pop arbitrary number of names from the stack. Here we use the technique from Example 3.3.

```
implicit {E,?L,t1} pop{env:push<E,?L,t1>}:E {
  return env pop;
}
```

Next, we define a super class for all states.

```
class State<K> {
  var next:K
  var b:Builder
  def State{n:K,b0:Builder} { next = n; b = b0; }
}
```

Next, we define the class that represents the first stack symbol.

```
class Ee<K,t,E> extends State<K> {
  def Ee{n:K,b0:Builder} { super{n,b0}; }
```

In this class, we define four methods. Variable:

```
  def {?L,t1,E1} (l:?L):K where
    p1:E→push<E1,?L,t1>; p2:t1≡t {
    b append{" "+(l toString)};
    return next;
  }
```

Constants:

```
  def (c:t):K {
    b append{" "+(c toString)};
    return next;
  }
```

Opening bracket:

```
  def {t1} [{}:Ee<Ee<Er<K>,t1,E>,fun<t1,t>> {
    b append{"["};
    return new Ee{new Ee{new Er{next,b},b},b};
  }
```

λ:

```
  def {t1,?L,t2} λ{}:
    Ev<Ed<Ee<K,t2,push<E,?L,t1>>>,?L> where
    p1:t≡fun<t1,t2> {
    b append{"λ"};
    return new Ev{new Ed{new Ee{next,b},b},b};
  }
}
```

We define the class that represents the second stack symbol.

```
class Er<K> extends State<K> {
  def Er{n:K,b0:Builder} { super{n,b0}; }
  def ]{}:K { b append{"]"}; return next; }
}
```

We define the class that represents the third stack symbol.

```
class Ev<K,?L> extends State<K> {
  def Ev{n:K,b0:Builder} { super{n,b0}; }
  def (l:?L):K {
    b append{l toString};
    return next;
  }
}
```

We define the class that represents the fourth stack symbol.

```
class Ed<K> extends State<K> {
  def Ed{n:K,b0:Builder} { super{n,b0}; }
  def :{}:K { b append{":"}; return next; }
}
```

We define the class that represents the end of a STLC term.

```
class Bot {
  var b:Builder
  def Bot{b0:Builder} { b = b0; }
  def toString{}:String { return b toString; }
}
```

We define a utility class that is used in the user code.

```
class Utils {
  def {t} prog{}:Ee<Bot,t,emp> {
    var b:Builder = new Builder{};
    return new Ee{new Bot{b},b};
  }
}
```

Finally, we can write code that generates a string representing for some STLC term. We can define an embedding that maps λ to λ, dot to colon, parentheses to square brackets, and every variable x to `#`x. Clearly, this is a usable embedding according to Section 2.1. For example,

```
class Main {
  def main{}:void {
    var program:String;
    program = new Utils{}
      prog λ#x:λ#y:[#y #x] toString;
  }
}
```

4.2 Typed Abstract Syntax Tree Builder

Example 4.2. A typed abstract syntax tree (AST) builder for the STLC. To simplify the code, we consider only abstraction and variables here.

We reuse the following definitions from the previous example.

```
interface push<E,?L,t> {
  def pop{}:E
}
interface emp { }
```

```
interface fun<t1,t2> { }
implicit {E,?L,t1} pop{env:push<E,?L,t1>}:E {
  return env pop;
}
```

Next, we define the classes for objects representing AST nodes.

```
interface Term<E,t> { }
class Abs<E,?L,t1,t2> implements
  Term<E,fun<t1,t2>> {
  var l:?L
  var e:Term<push<E,?L,t1>,t2>
  def Abs{l0:?L,e0:Term<push<E,?L,t1>,t2>} {
    l=l0; e=e0;
  }
}
class Var<E,?L,t> implements Term<E,t> {
  var l:?L
  def Var{l0:?L} { l=l0; }
}
```

Next, we define the interface for functions.

```
interface Func<S,T> {
  def @{x:S}:T
}
```

Next, we define classes that represent an automaton that builds an AST. A state of this automaton is a triple consisting of a stack, an AST builder, and a position in a term of the EDSL. The classes representing stack symbols are almost the same as the previous example, with an extra type parameter `S` which represents the AST builder of the current state. Note that the field `n` which stores the next state has type `Func<S,K>`. The reason that it has type `Func<S,K>` instead of `K` is that we want to pass the AST builder from the current state to the next state.

```
class Ee<K,t,E,S> {
  var n:Func<S,K>
  var b:Func<Term<t,E>,S>
  def Ee{n0:Func<S,K>,b0:Func<Term<t,E>,S>} {
    n=n0; b=b0;
  }
  def {?L,t1,E1} (l:?L):K where
    p1:E→push<E1,?L,t1>; p2:t1≡t {
    return n app{b @{new Var{l}}};
  }
  def {t1,?L,t2} λ{}:Ev<Ed<Ee<K,t2,push<E,?L,t1
      >>>,?L,Func<Term<t2,push<E,?L,t1>>,S>> where
    p:Term<E,t1→t2>≡Term<E,t> {
    return new Ev{
      new Func{} {def@{b1}{
        return new Ed{Ee{n,b1}};
      }},
      new Func{} {def@{l}{
        return new Func{} {def@{e}{
          return b @{p{new Abs{l,e}}};
        }};
      }}
    };
  }
}
class Ev<K,?L,S> {
  var n:Func<S,K>
  var b:Func<?L,S>
  def Ev{n0:Func<S,K>,b0:Func<?L,S>} {
    n=n0; b=b0;
  }
  def (l:?L):K { return n @{b @{l}}; }
}
class Ed<K> {
  var n:K
  def Ed{n0:K} { n=n0; }
  def :():K { return n; }
}
```

We define a utility class that is used in the user code.

```
class Utils {
  def {t} prog{}:Ee<Term<emp,t>,t,emp,Term<emp,t>>
      {
    return new Ee{
      new Func{} {def@{e}{
        return e;
      }},
      new Func{} {def@{e}{
        return e;
      }}
    };
  }
}
```

Finally, we have the user code. The user code builds a typed AST representing a term of the EDSL and stores it in the variable `ast`.

```
class Main {
  def main{}:void {
    var ast:Term<fun<Int,fun<Int,Int>>,emp>;
    ast = new Utils{}
      prog λ#x:λ#y:#x;
  }
}
```

4.3 Embedded Query

Example 4.3. Embedded Query. This example demonstrates how to encode an EDSL which allows us to write queries like the following.

```
select from{list} where #genre=="classical"
```

First, we define an interface for classes with getter methods.

```
interface get<T,?L> {
  T get(?L l)
}
```

Next, we define the class `CD` whose instances are used to store CD informations.

```
class CD implements
  get<String,#title>,
  get<String,#genre>,
  get<String,#artist> {
  var title:String
  var genre:String
  var artist:String
  def get{l:#title}:String { return title; }
  def get{l:#genre}:String { return genre; }
  def get{l:#artist}:String { return artist; }
}
```

Next, we add subtyping to the logic of implicit parameters.

```
order get<T,?L> {}
order Comparable<T> {}
implicit u{c:CD}:get<String,#title> { return c; }
implicit u{c:CD}:get<String,#genre> { return c; }
implicit u{c:CD}:get<String,#artist> { return c; }
implicit u{s:String}:Comparable<String> {
  return s;
}
```

Next, we define the interfaces representing the state of an automaton that accepts the embedded query EDSL. Here we are showing the interface only. The `from` clause:

```
interface Eselect {
  def {t} from{l:Collection<t> onerror
    ("select from a non collection")}:Ewhere<t>
}
```

The `where` clause:

```
interface Ewhere<t> { def where{}:Ecrit<t> }
```

A criterion in the `where` clause:

```
interface Ecrit<t> {
  def {t1} (l:?L):Ecmp<Eexp<Bot,t1,t>> where
    p:t→get<t1,?L> onerror
      (t+" does not contain property "+?L);
    p2:t1→Comparable<t1> onerror
      ("Uncomparable type of property "+?L+":"+t1)
}
interface Ecmp<K> {
  def =={}:K
  def !={}:K
  def <{}:K
  def >{}:K
  def <={}:K
  def >={}:K
}
interface Eexp<K,t1,t> {
  def (l:t onerror
    ("type mismatch")):K
  def (l:?L):K where
    p:t→get<t1, ?L> onerror
      (t+" does not contain property "+?L+":"+t1)
}
```

Next, we define a utility class as in the previous example.

```
class Utils {
  def select{}:Eselect { ... }
}
```

Finally, we have the user code which constructs a query that selects all CDs whose genre is `classical`. Unlike LINQ, no preprocesser is needed to translate the user code to a desugared form. Here, we also demostrate a technique that can be used to hide the initial object creation expression.

```
class Main extends Utils {
  def main{}:void {
    var lst:List<CD>;
    ...
    var q:Eprog;
    q = select from{lst} where #genre=="classical";
  }
}
```

5. FE: A core calculus for Eria

5.1 Internal Language

5.1.1 Syntax

The syntax of the internal language of FE is given in Figure 2. In order to make it easier to understand, we follow FGJ's conventions instead of directly using Eria's concrete syntax.

We focus on new features of Eria, by making a few simplifications in FE. We do not model fields, interfaces, bounds on type variables, method overloading, or constructors that take in parameters in the core calculus. FE has the following new language constructs (which will be explained in the following paragraphs) that are not in FGJ: label types A, implicit parameters types C, implicit functions i, implicit arguments p, implicit function definitions I, empty label ε, and appending $e\#l$ (a label character l to an expression e).

We define metavariables in a similar manner as FGJ. The metavariables D and E range over class names; m and n range over method names; x ranges over variables; e ranges over expressions; L ranges over class declarations; M ranges over method declarations; X, Y, and Z range over type variables; W ranges over nonlabel type variables; $?X$ ranges over label type variables; S, T, U, and V range over types; N, P, and Q range over nonvariable, nonlabel types; A and B range over label types; l ranges over label characters; p ranges over implicit arguments; C ranges over implicit parameter types; i ranges over implicit function names; and I ranges over implicit function definitions. We assume that the set of variables includes the special variable `this`, which is implicitly bound in every method definition. We denote an empty sequence by $\bullet$. We abbreviate $M_1\ \ldots\ M_n$ to $\overline{M}$ and $X_1, \ldots, X_n$ to $\overline{X}$, and similarly for other metavariables, with an exception that we abbreviate $C_1 \oplus \ldots \oplus C_n$ to $\overline{C}$, where $\oplus$ may be a comma or a semicolon. We denote the length of a sequence $\overline{M}$ by $|\overline{M}|$. We denote the concatenation of sequences with comma, with the exception that sequences of implicit parameters (types) can be concatenated with either a comma or a semicolon. We abbreviate operations on pairs by writing $\overline{T}\ \overline{x}$ for $T_1\ x_1, \ldots, T_n\ x_n$, etc. We assume that sequences of (implicit) parameter names and type variables contain no duplicate names.

There are two groups of types. The first group T are types that may occur in a definition or a term. The second group C are implicit parameter types. There are two kinds of implicit parameter types, equality types $S \equiv T$ and first order function types $(S) \to T$. We allow $(S) \to T$ to be abbreviated as $S \to T$. In typing rules, τ ranges over the union of T and C.

`class` $D\langle\overline{X}\rangle \lhd N\{\overline{M}\}$ introduces a class named D with supertype N and type parameters $\overline{X}$. The type variables that appear in N should also appear in $\overline{X}$. The class has methods $\overline{M}$. If a method of D overrides (has the same parameter types as) some method with the same name that is present in N, they should have the same return type and implicit parameter types.

$\langle\overline{X}\rangle\ T\ m(\overline{T}\ \overline{x} \mid \overline{C}\ \overline{y})\{$`return` $e;\}$ introduces a method named m with return type T, parameters $\overline{x}$ of types $\overline{T}$, and implicit parameters $\overline{y}$ of types $\overline{C}$. The type variables that appear in the return type or the parameter types must be in $\overline{X}$ or type parameters of the enclosing class definition. The body of the method is the statement `return` e;. The variables $\overline{x}$, $\overline{y}$ and the special variable `this` are bound in e.

`implicit` $\langle\overline{X}\rangle\ S\ i(\overline{T}\ \overline{x})\{$`return` $e;\}$ introduces an implicit function named i with parameters $\overline{x}$ of type $\overline{T}$ and return type S. The type variables that appear in S and $\overline{T}$ must appear in $\overline{X}$. The body of the implicit function is similar to that of a method, with the statement `return` e;. The variables $\overline{x}$ are bound in e.

`order` $D\ \langle\overline{X}\rangle\ \{\overline{X}\}$ defines the ord list for type constructor D. (Section 5.2.3)

There are three kinds of terms in Eria, the p-terms, the q-terms, and the e-terms. e-terms are just regular terms. ε introduces the empty label. $e\#l$ appends a label character

Types	T	$\rightarrow$	$X \mid N \mid A$
	X	$\rightarrow$	$W \mid ?X$
	N	$\rightarrow$	$D\langle\overline{T}\rangle$
	A	$\rightarrow$	$\epsilon \mid ?X\#l \mid A\#l$
	C	$\rightarrow$	$S \equiv T \mid \overline{T} \rightarrow T$
	τ	$\rightarrow$	$T \mid C$
Definitions	L	$\rightarrow$	$\texttt{class}\ D\langle\overline{X}\rangle \lhd N\{\overline{M}\}$
	M	$\rightarrow$	$\langle\overline{X}\rangle\ T\ m(\overline{T}\ \overline{x} \mid \overline{C}\ \overline{x})\{\texttt{return}\ e;\}$
	I	$\rightarrow$	$\texttt{implicit}\ \langle\overline{X}\rangle\ T\ i(\overline{S}\ \overline{x})\{\texttt{return}\ e;\}$
	O	$\rightarrow$	$\texttt{order}\ D\langle\overline{X}\rangle\ \{\overline{X}\}$
Terms	q	$\rightarrow$	$x \mid i\langle\overline{T}\rangle(\overline{q})$
	p	$\rightarrow$	$(\overline{x})q$
	e	$\rightarrow$	$\varepsilon \mid e\#l \mid x \mid \texttt{new}\ N() \mid e.m\langle\overline{T}\rangle(\overline{e} \mid \overline{p})$
			$\mid i\langle\overline{T}\rangle(\overline{e}) \mid x(\overline{e}) \mid x \triangleright (\overline{e})$
Environments	Γ	$\rightarrow$	$\overline{x} : \overline{T}$
Values	a	$\rightarrow$	$\varepsilon \mid a\#l$
	w	$\rightarrow$	$new\ N() \mid a$

Figure 2. The Internal Language

l to a label e. x introduces a variable. $\texttt{new}\ N()$ creates an object of type N, $e.m\langle\overline{T}\rangle(\overline{e} \mid \overline{p})$ invokes a method m with type arguments $\overline{T}$, arguments $\overline{e}$, and implicit arguments $\overline{p}$ at object e. $i\langle\overline{T}\rangle(\overline{e})$ applies an implicit function i to type arguments $\overline{T}$ and arguments $\overline{e}$. $x(\overline{e})$ applies an implicit parameter x with a function type to arguments $\overline{e}$. $x \triangleright (e)$ applies an implicit parameter of an equality type $S \equiv T$ as a term of type $S \rightarrow T$ to argument e. Implicit arguments are p-terms, which are abstractions of the form $(\overline{x})q$, where the variables $\overline{x}$ are bounded in q. q-terms include variable x and application $i\langle\overline{T}\rangle(\overline{q})$ of an implicit function i to type arguments $\overline{T}$ and arguments $\overline{q}$.

5.1.2 Auxiliary Definitions

We assume a fixed class table DT, a mapping from class names D to class definitions L, a fixed implicit function table IT, a mapping from implicit function names i to implicit function definitions I, and a fixed order table OT, a mapping from class names D to lists of integers. A program is a quadruple (DT, IT, OT, e) of a class table, an implicit function table, an order table, and an expression. We have a predefined class `object` whose definition does not appear in the class table. We also have a predefined implicit function $\texttt{implict}\ \langle T\rangle\ \texttt{id}(T\ x)\{\texttt{return}\ x;\}$. We assume that the class table DT satisfy the sanity conditions similar to FGJ, which is not reiterated here.

We need a few auxiliary definitions for the typing rules, as shown in Figure 3. The rules are very similar to FGJ. $m \notin \overline{M}$ stands for that the method definition of m is not included in $\overline{M}$. Application of type substitution $[\overline{T}/\overline{X}]$ to a term e, written $[\overline{T}/\overline{X}]e$, is defined as simultaneous substitution of type variables $\overline{X}$ by types $\overline{T}$. The type of method invocation m at type N, written $\mathsf{mtype}(m, N)$, is a type of the form $\langle\overline{X}\rangle\ \overline{U} \mid \overline{C} \rightarrow U$, in which the variables $\overline{X}$ are bound in $\overline{U}$, U, and $\overline{C}$. The body of method invocation m at type N, written $\mathsf{mbody}(m, N)$, is $(\overline{x})e$, where $\overline{x}$ is a sequence of parameters bound in e which is an e-term.

Method type lookup

$$\frac{\texttt{class}\ C\langle\overline{X}\rangle \lhd N\{\overline{M}\} \qquad \langle\overline{Y}\rangle\ U\ m(\overline{U}\ \overline{x} \mid \overline{C}\ \overline{y})\ \{\ldots\} \in \overline{M}}{\mathsf{mtype}(m, C\langle\overline{T}\rangle) = [\overline{T}/\overline{X}](\langle\overline{Y}\rangle\overline{U} \mid \overline{C} \rightarrow U)}$$

$$\frac{\texttt{class}\ C\langle\overline{X}\rangle \lhd N\{\overline{M}\} \qquad m \notin \overline{M}}{\mathsf{mtype}(m, C\langle\overline{T}\rangle) = \mathsf{mtype}(m, [\overline{T}/\overline{X}]N)}$$

Method body lookup

$$\frac{\texttt{class}\ D\langle\overline{X}\rangle \lhd N\{\overline{M}\} \quad \langle\overline{Y}\rangle\ U\ m(\overline{U}\ \overline{x} \mid \overline{C}\ \overline{y})\ \{\texttt{return}\ e;\} \in \overline{M}}{\mathsf{mbody}(m\langle\overline{V}\rangle, D\langle\overline{T}\rangle) = (\overline{x})[\overline{T}/\overline{X}, \overline{V}/\overline{Y}]e}$$

$$\frac{\texttt{class}\ D\langle\overline{X}\rangle \lhd N\{\overline{M}\} \qquad m \notin \overline{M}}{\mathsf{mbody}(m\langle\overline{V}\rangle, D\langle\overline{T}\rangle) = \mathsf{mbody}(m\langle\overline{V}\rangle, [\overline{T}/\overline{X}]N)}$$

Overriding

$$\frac{\mathsf{mtype}(m, N) = \langle\overline{Z}\rangle\overline{U} \mid \overline{C'} \rightarrow U_0 \quad T_0, \overline{T}, \overline{C} = [\overline{Y}/\overline{Z}](U_0, \overline{U}, \overline{C'})}{\mathsf{override}(m, N, \langle\overline{Y}\rangle\overline{T} \mid \overline{C} \rightarrow T_0)}$$

Implicit function type lookup

$$\frac{\texttt{implicit}\ \langle\overline{X}\rangle\ T\ i(\overline{S}\ \overline{x})\ \{\ldots\}}{\mathsf{itype}(i) = \langle\overline{X}\rangle\overline{S} \rightarrow T}$$

Implicit function body lookup

$$\frac{\texttt{implicit}\ \langle\overline{X}\rangle\ T\ i(\overline{S}\ \overline{x})\ \{\texttt{return}\ e;\}}{\mathsf{ibody}(i\langle\overline{V}\rangle) = (\overline{x})[\overline{V}/\overline{X}]e}$$

Figure 3. Auxiliary Functions

The type of implicit function i, written $\mathsf{itype}(i)$, is a type of the form $\langle\overline{X}\rangle U$ or the form $\langle\overline{X}\rangle\overline{U} \rightarrow U$, in which the variables $\overline{X}$ are bound in $\overline{U}$ and U. The body of implicit function i, written $\mathsf{ibody}(i)$, is $(\overline{x})q$, where $\overline{x}$ is a sequence of parameters bound in q which is a q-term. In a type of the form $\langle\overline{X}\rangle\overline{U} \rightarrow U$ or $\langle\overline{X}\rangle\overline{U} \mid \overline{C} \rightarrow U$, $\overline{U}$ or $\overline{U} \mid \overline{C}$ is called the domain, and U is called the codomain.

5.1.3 Typing

A typing environment Γ is a finite mapping from variables to types, written $\overline{x} : \overline{T}$. (Figure 2)

A typing environment Γ is well-formed if all types appearing in Γ are well-formed. The type `object`, a type variable, or a label type is always well-formed. If the definition of a class D begins with $\texttt{class}\ D\langle\overline{X}\rangle$, then a type like $D\langle\overline{T}\rangle$ is well-formed if $\overline{T}$ are well-formed. A first-order function type is well-formed if its domain and codomain are well-formed. An equality type is well-formed if its left hand side and right hand side are both well-formed. A substitution is

Subtyping

$$T <: T \qquad \frac{S <: T \qquad T <: U}{S <: U} \qquad \frac{\texttt{class}\ D\langle \overline{X}\rangle \lhd N\{\ldots\}}{D\langle \overline{T}\rangle <: [\overline{T}/\overline{X}]N}$$

Figure 4. Subtyping

well-formed if it substitutes a regular type variable with a well-formed typed and a label variable by a label type.

Judgments include subtyping $S <: T$ and term typings $\Gamma \vdash^I p : \tau$, $\Gamma \vdash^I q : \tau$, and $\Gamma \vdash e : T$. A sequence of judgments of the same kind is abbreviated in the same way as FGJ. For example, $S_1 <: T_1, \ldots S_n <: T_n$ to $\overline{S} <: \overline{T}$.

The subtyping relations $S <: T$ are defined in Figure 4. Type parameters are invariant w.r.t. subtyping. For example, in order for $D\langle S\rangle <: D\langle T\rangle$, we must have $S = T$.

The typing rules for p-terms and q-terms are defined in the first section of Figure 5. The (ITVAR) rule says that a variable has the type assigned by the typing environment. The (ITAPP) rule models application. The (ITABS) rule models monomorphic abstraction. The (ITEQ) rule says that $\texttt{id}\langle T\rangle$ can be used as an implicit argument of equality type $T \equiv T$. In fact the only inhabited equality types in FE are types whose left hand side and right hand side are definitional equal, and the only value inhabiting an equality type $T \equiv T$ is $\texttt{id}\langle T\rangle$.

The typing rules for e-terms are shown in the second section of Figure 5. (The differences between FE and FGJ in the typing rules are highlighted.) The (FTVAR) rule says that the type of a variable is that declared in the typing environment. The (FTEMP) rule says that an empty label ε has type ϵ. The (FTAPND) rule says that appending a label character l to an expression e of label type A results in an expression of type $A\#l$. The (FTINV) rule says that for a method invocation in a method chain, the method invocation should satisfy the common requirements for parameter types and return type, as well as that $\overline{p}$ should have the types $\overline{C}$. The (FTAPPIMP) rule says that to apply an implicit function to some expressions, the types of the expressions should be subtypes of the domain of the type of the implicit function. The (FTAPPFUNC) rule says that to apply an implicit parameter of a function type to some expressions, the application should satisfy the requirement that the types of the expressions are subtypes of the domain of the type of the implicit parameter. The (FTAPPEQ) rule says that an implicit parameter x of type $S \equiv T$ can be used as a function of type $S \to T$.

The typing rules for classes, methods, and implicit functions are shown in the rest of Figure 5. The (FTMETH) rule says that a method is well-typed if given its parameter types, the type of the enclosing class, and implicit parameter types, the return value has a type that is a subtype of the return type. The (FTCLASS) rule says that a class is well-typed if its methods are well-typed. The (FTIMP) rule says that an implicit function is well-typed if given its parameter types, the return value has a type that is a subtype of the return type.

5.1.4 Computation

The rules for value and computation is given in Figure 7. The rules are similar to those of FGJ. We leave out the congruence rules, which are standard, because of space limit.

p-term and q-term typing $\Gamma \vdash^I q : \tau$ and $\Gamma \vdash^I p : \tau$

$$\Gamma \vdash x : \Gamma(x) \quad \text{(ITVAR)}$$

$$\frac{\mathsf{itype}(i) = \langle \overline{X}\rangle \overline{T} \to T \qquad \Gamma \vdash \overline{q} : [\overline{V}/\overline{X}]\overline{T}}{\Gamma \vdash i\langle \overline{V}\rangle(\overline{q}) : [\overline{V}/\overline{X}]T} \quad \text{(ITAPP)}$$

$$\frac{\Gamma, \overline{x} : \overline{T} \vdash q : T}{\Gamma \vdash (\overline{x})q : (\overline{T}) \to T} \quad \text{(ITABS)}$$

$$\Gamma \vdash^I \texttt{id}\langle T\rangle : T \equiv T \quad \text{(ITEQ)}$$

e-term typing $\Gamma \vdash e : T$

$$\Gamma \vdash x : \Gamma(x) \quad \text{(FTVAR)} \qquad \Gamma \vdash \varepsilon : \epsilon \quad \text{(FTEMP)}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e\#l : A\#l} \quad \text{(FTAPND)} \qquad \Gamma \vdash \texttt{new}\ N() : N \quad \text{(FTNEW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \qquad \mathsf{mtype}(m, T_0) = \langle \overline{Y}\rangle \overline{U} |\ \overline{C} \to U \\ \Gamma \vdash \overline{e} : \overline{S} \qquad \bullet \vdash^I \overline{p} : [\overline{V}/\overline{Y}]\overline{C} \qquad \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}\end{array}}{\Gamma \vdash e_0.m\langle \overline{V}\rangle(\overline{e}|\ \overline{p}) : [\overline{V}/\overline{Y}]U} \quad \text{(FTINV)}$$

$$\frac{\begin{array}{c}\mathsf{itype}(i) = \langle \overline{X}\rangle \overline{U} \to U \\ \Gamma \vdash \overline{e} : \overline{S} \qquad \overline{S} <: [\overline{T}/\overline{X}]\overline{U}\end{array}}{\Gamma \vdash i\langle \overline{T}\rangle(\overline{e}) : [\overline{T}/\overline{X}]U} \quad \text{(FTAPPIMP)}$$

$$\frac{\Gamma(x) = \overline{U} \to U \qquad \Gamma \vdash \overline{e} : \overline{S} \qquad \overline{S} <: \overline{U}}{\Gamma \vdash x(\overline{e}) : U} \quad \text{(FTAPPFUNC)}$$

$$\frac{\Gamma(x) : U \equiv T \qquad \Gamma \vdash e : S \qquad S <: U}{\Gamma \vdash x \triangleright (e) : T} \quad \text{(FTAPPEQ)}$$

Method typing

$$\frac{\begin{array}{c}\overline{x} : \overline{T}, \overline{y} : \overline{C}, \texttt{this} : D\langle \overline{X}\rangle \vdash e_0 : S \\ S <: T \qquad \texttt{class}\ D\langle \overline{X}\rangle \lhd N\{\ldots\} \\ \mathsf{override}(m, N, \langle \overline{Y}\rangle \overline{T} |\ \overline{C} \to T)\end{array}}{\langle \overline{Y}\rangle\ T\ m(\overline{T}\ \overline{x} |\ \overline{C}\ \overline{y})\{\texttt{return}\ e_0;\}\ \mathsf{ok\ in}\ D\langle \overline{X}\rangle} \quad \text{(FTMETH)}$$

Class typing

$$\frac{\overline{M}\ \mathsf{ok\ in}\ D\langle \overline{X}\rangle}{\texttt{class}\ D\langle \overline{X}\rangle \lhd N\{\overline{M}\}\ \mathsf{ok}} \quad \text{(FTCLASS)}$$

Implicit function typing

$$\frac{\overline{X}; \overline{x} : \overline{S} \vdash e : T_0 \qquad \overline{X} \vdash T_0 <: T}{\texttt{implicit}\ \langle \overline{X}\rangle\ T\ i(\overline{S}\ x)\{\texttt{return}\ e;\}\ \mathsf{ok}} \quad \text{(FTIMP)}$$

Figure 5. Typing Rules

Computation $e \longrightarrow e'$

$$\frac{\mathsf{mbody}(m\langle\overline{V}\rangle, N) = (\overline{x} \mid \overline{y})e_0}{(\texttt{new}\ N()).m\langle\overline{V}\rangle(\overline{d} \mid \overline{p}) \longrightarrow [\overline{d}/\overline{x}, \overline{p}/\overline{y}, \texttt{new}\ N()/\texttt{this}]e_0} \quad (\text{FRINV})$$

$$\frac{\mathsf{ibody}(i\langle\overline{V}\rangle) = (\overline{x})e_0}{i\langle\overline{V}\rangle(\overline{d}) \longrightarrow [\overline{d}/\overline{x}]e_0} \quad (\text{FRAPPIMP})$$

$$(\overline{x})q_0(\overline{d}) \longrightarrow [\overline{d}/\overline{x}]q_0 \quad (\text{FRAPPABS})$$

$$\mathsf{id}\langle\overline{T}\rangle \triangleright (d) \longrightarrow d \quad (\text{FREQ})$$

Figure 6. Computation

5.1.5 Properties

We can define a compiling function $\| - \|$ that transform an FE program to an FGJ program so that for every FE type in the FE program there is a unique FGJ type in the FGJ program. Because of space limit, we demonstrate the compiling function using a few examples. We assume that new class names are fresh, omit empty constructors, and abbreviate `return` (T)`new object`$()$; to $\bot$.

For example, labels can be compiled as follows. We defined

$$\texttt{class}\ \epsilon\{\}$$

for the empty label and

$$\texttt{class}\ l\langle T\rangle\{T\ f; l(T\ f0)\{f = f0;\}\}$$

for each label character l. The empty label type can be compiled to

$$\epsilon$$

and every label type $\#l_1 \ldots l_n$ can be compiled to

$$l_n\langle \ldots \langle l_1\langle\epsilon\rangle\rangle \ldots\rangle$$

The empty label can be compiled to

$$\texttt{new}\ \epsilon()$$

and every label $\#l_1 \ldots l_n$ can be compiled to

$$\texttt{new}\ l_n\langle \ldots \langle l_1\langle\epsilon\rangle\rangle \ldots\rangle(\ldots \texttt{new}\ l_1\langle\epsilon\rangle(\texttt{new}\ \epsilon()) \ldots)$$

For another example, equality types can be compiled as follows. For equality types, we define

$$\texttt{class}\ eq\langle S, T\rangle\{T\ app(S\ x)\{\bot\}\}$$

and

$$\texttt{class}\ id\langle T\rangle \lhd eq\langle T, T\rangle\{T\ app(T\ x)\{\texttt{return}\ x;\}\}$$

If an implicit parameter x has type $S \equiv T$, then $x \triangleright (e)$ can be compiled to

$$x.app(\|e\|)$$

It is obvious that the compiling function in the example is an injection from FE terms and types to FGJ terms and types. Using this compiling function, the following properties follow from the properties of FGJ.

Terms $e \rightarrow \varepsilon \mid e\#l \mid x \mid \texttt{new}\ D() \mid e.m(\overline{e}) \mid x(\overline{e}) \mid x \triangleright (e)$

Figure 7. The External Language

Theorem 5.1. (Preservation) *If* $\Gamma \vdash e : T$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : T'$ *for some* T' *such that* $T' <: T$.

Theorem 5.2. (Progress) *Suppose that* e *is a well-typed* e*-term.*

1. *If* e *includes* `new` $N_0(\overline{e}).m\langle\overline{V}\rangle(\overline{d})$ *as a subterm, then* $\mathsf{mbody}(m\langle\overline{V}\rangle, N_0) = (\overline{x})e_0$ *and* $|\overline{x}| = |\overline{d}|$ *for some* $\overline{x}$ *and* e_0.
2. *If* e *includes* $i\langle\overline{V}\rangle(\overline{d})$ *as a subterm, then* $\mathsf{ibody}(i\langle\overline{V}\rangle) = (\overline{x})e_0$ *and* $|\overline{x}| = |\overline{d}|$ *for some* $\overline{x}$ *and* e_0.
3. *If* e *includes* $(\overline{x})q(\overline{d})$ *as a subterm, then* $|\overline{x}| = |\overline{d}|$.

Theorem 5.3. (Type soundness) *If* $\bullet \vdash e : T$ *and* $e \longrightarrow^* e'$ *with* e' *a normal form, then* e' *is a value* w *with* $\bullet \vdash w : S$ *and* $S <: T$.

5.2 External Language

5.2.1 Syntax

The only difference between the internal language, which is the language in Figure 2 and the external language is that the external language does not have type parameters, implicit arguments, or implicit function applications in the terms, as shown in Figure 7.

5.2.2 Implicit Parameter Inference

Implicit parameter inference is the key feature of Eria that enables usable encoding of typed object languages. In contrast to type inference which infers a type given an untyped term, implicit parameter inference infers a term given a type. The term inferred has a type that is an instance of the given type obtained by applying some substitution to the given type.

We let θ range over substitutions, and denote composition of substitutions θ_1, θ_2 by $\theta_1 \circ \theta_2$, defined as $\theta_1 \circ \theta_2(x) = \theta_1(\theta_2(x))$.

Formally speaking, implicit parameter inference infers an argument p and a substitution θ given implicit parameter type C and a typing environment Γ, written $\Gamma \vdash^F_\theta p : C$. We will show in Theorem 5.5 that the inferred argument p has the following typing $\theta\Gamma \vdash^I p : \theta C$. The algorithm is given in Figure 8.

These rules are explained below by analogy with Prolog. Variables and their types in a typing environment Γ correspond to local facts (assumptions). They do not correspond to rules because they do not have function types. For example, a variable x with type $\texttt{nat}\langle\texttt{zero}\rangle$ in a typing environment may correspond to a local fact that `zero` is a natural number. The implicit functions and their types correspond to global rules and facts. For example, an implicit function with type $() \rightarrow \texttt{nat}\langle\texttt{zero}\rangle$ corresponds to a global fact (where a fact can be thought of as a rule with an empty rule body), and an implicit function with type $\texttt{nat}\langle\texttt{N}\rangle \rightarrow \texttt{nat}\langle\texttt{succ}\langle\texttt{N}\rangle\rangle$ corresponds to a rule.

In the following rules, the function mgu computes the most general unifier.

Implicit parameter inference $\Gamma \vdash^F_\theta q : C$ and $\Gamma \vdash^F_\theta p : C$

$$\frac{\theta = \mathsf{mgu}(S, T)}{\Gamma \vdash^F_\theta \mathtt{id}\langle\theta S\rangle : S \equiv T} \quad (\text{IIEQ})$$

$$\frac{T \notin \overline{T'} \qquad \theta = \mathsf{mgu}(\Gamma(x), T)}{\Gamma \vdash^F_\theta x : T} \quad (\text{IIVAR})$$

$$\frac{\begin{array}{c}\mathsf{itype}(i) = \langle\overline{X}\rangle(\overline{U}) \to U \\ \theta_0 = \mathsf{mgu}(U, T) \qquad \theta_0\Gamma \vdash^F_\theta \overline{q} : \theta_0\overline{U}\end{array}}{\Gamma \vdash^F_{\theta\circ\theta_0} i\langle\theta\theta_0\overline{X}\rangle(\overline{q}) : T} \quad (\text{IIAPP})$$

$$\frac{\Gamma, \overline{x} : \overline{T} \vdash^F_\theta q : T}{\Gamma \vdash^F_\theta (\overline{x})q : (\overline{T}) \to T} \quad (\text{IIABS})$$

$$\Gamma \vdash^F_{[]} \bullet : \bullet \quad (\text{IIEMPTYSEQ})$$

$$\frac{\Gamma \vdash^F_{\theta_0} \overline{p} : \overline{T} \qquad \theta_0\Gamma \vdash^F_\theta p : \theta_0 T}{\Gamma \vdash^F_{\theta\circ\theta_0} \overline{p}, p : \overline{T}, T} \quad (\text{IICOMMA})$$

$$\frac{\Gamma \vdash^F_{\theta_0} \overline{p} : \overline{T} \qquad \mathsf{cut} \qquad \theta_0\Gamma \vdash^F_\theta p : \theta_0 T}{\Gamma \vdash^F_{\theta\circ\theta_0} \overline{p}; p : \overline{T}; T} \quad (\text{IISEMICOLON})$$

Figure 8. Implicit parameter inference

The rules is a variant of the SLD resolution. Like the SLD resolution, the rules maintain a list of goals and generate a substitution. Unlike the standard SLD resolution, these rules also construct proof terms from the goals. Given an implicit parameter type $(\overline{T}) \to T$, the (IIABS) rule adds the types $\overline{T}$ into the typing environment, which corresponds to adding $\overline{T}$ to a list of local facts and set T as the initial goal. The (IIAPP) rule picks a rule, the rule head of which unifies with one of the goals, and replaces that goal with the rule body of the rule picked, where the $\not\subset$ denotes that a list is not contained in another list regardless of order of elements. The (IIVAR) rule picks a local fact that unifies with one of the goals, and deletes that goal. The (IIDELETE) rule deletes a repeated goal from the list of goals. The (IIEQ) rule unifies the two sides of an goal, if the goal is an equality type. Given a goal, the obtained substitution θ corresponds to values of the type variables that make the goal succeed, and the term p is a proof that the values do do so.

The (IICOMMA) rule and the (IISEMICOLON) rule deal with consecutive goals in a query that allow and do not allow backtracking, respectively. The premises of these two rules should be read as searching. The cut in the (IISEMICOLON) rule prevents the rule from backtracking, similar to the "cut" in Prolog.

The order is apparently very important, as difference search order would result in different constructed term. We perform depth-first search and when multiple rules are applicable. We always select the first applicable rule to the leftmost goal in the following order: first (IIVAR) in the order that variables are arranged in the typing environment and then (IIAPP) in the order that the implicit functions are defined.

Order $<_{sl}$

$$\frac{|c| < |d|}{c <_{sl} d} \qquad \frac{|c| = |d| \qquad c_1 <_l d_1}{c <_{sl} d}$$

$$\frac{|c| = |d| \qquad c_1 = d_1 \qquad c_2 \ldots c_{|c|} <_{sl} d_2 \ldots d_{|d|}}{c <_{sl} d}$$

Order string $\mathsf{str}(\tau)$

$$\mathsf{str}(X) = X \qquad \mathsf{str}(A) = A$$

$$\frac{\mathtt{order}\ D\langle\overline{X}\rangle\{X_{i_1}, \ldots, X_{i_k}\}}{\mathsf{str}(D\langle\overline{T}\rangle) = D\mathsf{str}(T_{i_1}) \ldots \mathsf{str}(T_{i_k})}$$

Order $<_{sig}$

$$\frac{\mathsf{str}(T_1) <_{sl} \mathsf{str}(T_2)}{T_1 <_{sig} T_2}$$

Figure 9. Orders

By induction on the derivation of $\Gamma \vdash^F_{\theta_0} p : C$ and $\Gamma \vdash^F_\theta q : C$, we can prove the following.

Lemma 5.4. *If $\Gamma \vdash^F_{\theta_0} p : C$, then $\Gamma \vdash^F_{\theta_1\circ\theta_0} \theta_1 p : C$ for all substitution θ_1; if $\Gamma \vdash^F_{\theta_0} q : C$, then $\Gamma \vdash^F_{\theta_1\circ\theta_0} \theta_1 q : C$ for all substitution θ_1.*

Theorem 5.5. (Soundness of implicit parameter inference) *If $\Gamma \vdash^F_\theta p : C$, then $\theta\Gamma \vdash^I p : \theta C$.*

We do not need a completeness theorem. Because of (IISEMICOLON), completeness is not guaranteed.

5.2.3 Decidable Implicit Parameter Inference

We discuss how to make $\Gamma \vdash^F_\theta p \triangleright C$ decidable in this subsubsection. In general, SLD resolution on Horn clauses is semidecidable: it may not terminate if no solution can be found, which is undesirable for a type checker. Finding a decidable implicit parameter inference algorithm corresponds to finding a decidable logic and its decision procedure.

The approach Eria takes is based on a well-founded order on types.

Let us assume for now that we have a well-founded order $<_l$ on symbols such as type constructors and variables. We can define another well-founded order on types: the sizelexicographic order $<_{sl}$, defined in the Figure 9. When computing the order of types, we view types as a sequence of symbols, or flatterms, by removing all angle brackets and commas from its representation. For example, the type $D\,\langle E_1, E_2\rangle$ is written DE_1E_2. We denote flatterms by $c, d, \ldots$. A subscript i selects the ith symbol in a flatterm. Intuitively, the sizelexicographic order orders flatterms first by size and then by dictionary order with regard to $<_l$. This allows the ordering such as $DX <_{sl} EX$ given $D <_l E$, and $DX <_{sl} EXY$.

To add a little more flexibility to $<_{sl}$, we allow type parameters to a type constructor to be ignore or reordered in the order $<_{sig}$. Suppose that for an $n$ary type construc-

Decidable implicit parameter inference $\Gamma \vdash^{FD}_{\theta} q : C$ and $\Gamma \vdash^{FD}_{\theta} p : C$

$$\frac{T \notin \overline{T'} \qquad \Gamma(x) = \theta T}{\Gamma \vdash^{FD}_{\theta} x : T} \quad (\text{DIIVAR})$$

$$\frac{\mathsf{itype}(i) = \langle \overline{X} \rangle() \to U \qquad U = \theta_0 T}{\Gamma \vdash^{FD}_{\theta\circ\theta_0} i\langle\theta\overline{X}\rangle() : T} \quad (\text{DIIAPPFACT})$$

$$\frac{\begin{array}{c}\mathsf{itype}(i) = \langle \overline{X} \rangle(\overline{U}) \to U \qquad \overline{U} \neq \bullet \\ \theta_0 = \mathsf{mgu}(U, T) \qquad \theta_0\Gamma \vdash^{FD}_{\theta} \overline{q} : \theta_0\overline{U}\end{array}}{\Gamma \vdash^{FD}_{\theta\circ\theta_0} i\langle\theta\theta_0\overline{X}\rangle(\overline{q}) : T} \quad (\text{DIIAPPRULE})$$

Figure 10. Decidable Implicit parameter inference

tor D, we have a list of integers between 1 and n, written $\mathsf{ord}(D)$. Note that we use a list instead of a set to allow repetition. The string of a type $D\langle T_1, \ldots, T_n\rangle$, written $\mathsf{str}(D\langle T_1, \ldots, T_n\rangle)$, is defined as $D\mathsf{str}(T_{i_1}) \ldots \mathsf{str}(T_{i_k})$, where $\mathsf{ord}(D) = i_1, \ldots, i_k$. The order $<_{sig}$ is defined as $T_1 <_{sig} T_2$ if and only if $\mathsf{str}(T_1) <_{sl} \mathsf{str}(T_2)$. It is obvious that if $<_l$ is a well-founded order, then $<_{sig}$ is a well-founded order.

Next, we look at how to use $<_{sig}$ to guarantee termination of term inference. We make the following two restrictions. First, an implicit function that has at least one parameters should have type $(T_1, \ldots, T_n) \to T$ s.t. $\theta T <_{sig} \theta T_i$ for some integer $i \in [1, n]$ and all substitutions θ. Second, we modify the rules of implicit parameter inference to restrict where unification can occur. We denote the modified inference by $\Gamma \vdash^{FD}_{\theta} p : C$. We show only the rules that are modified in Figure 10.

In this version, the facts may not be instantiated but rules may be instantiated, unlike in $\Gamma \vdash^{F}_{\theta} p : C$ where both can be instantiated. Because of this distinction, we split the (IIAPP) rule into two rules: (DIIAPPRULE) and (DIIAPPFACT). The main modification is that in the rules (DIIAPPFACT) and (DIIVAR), the variables in goals can be instantiated but the variables in the fact can not be instantiated.

It is clear now how these restrictions guarantee the decidability of $\Gamma \vdash^{FD}_{\theta} p : C$. By the ordering requirement, in the rule (DIIAPPRULE), we have $T = \theta_0 T' <_{sig} \theta_0 S_i$ for some integer i, which means that every time the rule (DIIAPPRULE) is applied, we generate a goal that is strictly larger than one of the existing goals w.r.t. $<_{sig}$. On the other hand, we have finite number of local and global facts that can be used to reduce the number of goals, by either the rule (DIIAPPFACT) or the rule (DIIVAR). Since in the rules (DIIAPPFACT) and (DIIVAR), the variables in the fact can not be instantiated, there is a bound on the size of the goal that can be deleted. By the well-foundedness $<_{sig}$, the number of goal that can be deleted is bounded. If all goal are deleted, then the algorithm succeeds; if any of the goals goes over the bound, then it fails.

Theorem 5.6. *If* $\Gamma \vdash^{FD}_{\theta} p : C$, *then* $\Gamma \vdash^{F}_{\theta} p : C$.

Theorem 5.7. (Decidability of decidable implicit parameter inference) $\Gamma \vdash^{FD}_{\theta} p : C$ *is decidable.*

The order $<_l$ may be constructed as follows. We construct a directed graph. For each implicit function of type $(\overline{S}) \to T$ such that for all integer $k \in [1, |\overline{S}|]$, $|\mathsf{str}(\theta S_k)| \leq |\mathsf{str}(\theta T)|$ for some θ, we nondeterministically choose a k such that $|\mathsf{str}(\theta S_k)| = |\mathsf{str}(\theta T)|$ for all θ. If we can not find such a k, then inference fails. Otherwise, we choose the smallest n such that the nth symbol in $\mathsf{str}(S_k)$ differs from the nth symbol in $\mathsf{str}(T)$. If any of the two symbols is a variable, then try another k. Otherwise, we add an edge from the symbol in $\mathsf{str}(S_k)$ to the symbol in $\mathsf{str}(T)$. If the resulting graph is cyclic, then delete the edge and try another k. After all edges are added, we obtain $<_l$ by topological sorting on the graph. Although not complete, this algorithm works for all the examples in Section 4.

5.2.4 Type Inference

The algorithm is based on local type inference [30] with the addition that substitutions obtained from implicit parameter inference are also used to instantiate type variables. We do not elaborate here but list the rules for completeness.

Type inference is given by $\Gamma \vdash_{\theta} e : T \Rightarrow e'$, where the input is a term e in the external language, a typing environment Γ, and the output is a term e' in the internal language, a type T, and a substitution θ. The internal language term e' is well-typed and has the following typing $\theta\Gamma \vdash e' : \theta T$ if type inference succeeds. The algorithm is given in Figure 11.

The type inference algorithm invokes the subtyping constraint solver, which we denote by $\Gamma \vdash_{\theta} \overline{S} <: \overline{T}$, from local type inference. $\Gamma \vdash_{\theta} \overline{S} <: \overline{T}$ returns a substitution θ, given subtyping constraints $\overline{S} <: \overline{T}$ and typing environments Γ, such that $\theta\Gamma \vdash \theta\overline{S} <: \theta\overline{T}$.

6. Related Work

Object-Oriented Programming Eria is based on Java [11]. There are many similarity between Eria and Scala [26]. However, there are many features of Eria that are in Scala, as Eria is designed as a meta language for EDSLs while Scala is not, even thought it is easy to defined some EDSLs in Scala. The difference in design objectives results in some subtle differences between Eria and Java or Scala. Implicit parameters in Eria is very different from implicit parameters in Scala. Implicit parameter in Scala are inferred from a specific set of rules as defined in the Scala Language Specification [26]. In Eria, the argument is inferred from its type and types of implicit functions, which is one of the central ideas in Eria design for supporting complex EDSLs. The difference between implicit functions in Eria and views in Scala is subtler. Only one view can be involved in one application of implicit conversion and views are applied implicitly according to a set of rules as defined in the Scala Language Specification [26]. Multiple implicit functions can be combined and their application is controlled by the programmer.

The equality type for implicit parameters is inspired by a similar typing constraint proposed for $C\sharp$ [17]. Implicit parameter types differ from typing constraints in that they are not only constraints that need to be solved but also specifications from which implicit arguments are inferred.

Predicate dispatch [5, 7, 23, 27, 33] is a mechanism for determining the code to be executed upon a method invocation, by user defined predicates. Languages that support predicate dispatch usually have specialized logic that incorporates the OO concepts. In comparison, programmable in-

Type Inference $\Gamma \vdash_\theta e : T \Rightarrow e'$

$$\frac{T = \Gamma(x)}{\Gamma \vdash_{[]} x : T \Rightarrow x} \quad \text{(FIVAR)} \qquad \Gamma \vdash_{[]} \varepsilon : \epsilon \Rightarrow \varepsilon \quad \text{(FIEMP)}$$

$$\frac{\Gamma \vdash_\theta e : A \Rightarrow e'}{\Gamma \vdash_\theta e\#l : A\#l \Rightarrow e'\#l} \quad \text{(FIAPND)}$$

$$\frac{\texttt{class}\ D\langle\overline{X}\rangle\ \{\ldots\} \qquad \overline{Y}\ \text{fresh}}{\Gamma \vdash_\theta \texttt{new}\ D() : D\langle\overline{Y}\rangle \Rightarrow \texttt{new}\ D\langle\overline{Y}\rangle()} \quad \text{(FINEW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\theta_0} e_0 : T_0 \Rightarrow e_0' \\ \mathsf{mtype}(m, \theta_0 T_0) = \langle\overline{Y}\rangle\overline{U} \mid \overline{C} \to U \\ \theta_0\Gamma \vdash_{\theta_1} \overline{e} : \theta_0\overline{S} \Rightarrow \overline{e'} \\ \theta_1\theta_0\Gamma \vdash_{\theta_2} \theta_1\theta_0\overline{S} <: \theta_1\theta_0\overline{U} \qquad \bullet \vdash^F_\theta \overline{p} : \theta_2\theta_1\theta_0\overline{C}\end{array}}{\begin{array}{c}\Gamma \vdash_{\theta\circ\theta_1\circ\theta_0\circ\theta_2} e_0.m(\overline{e}) : U \Rightarrow e_0''.m\langle\overline{T}\rangle(\overline{e''} \mid \overline{p})) \\ \text{where } e_0'' = \theta\theta_2\theta_1 e_0', \overline{T} = \theta\theta_2\theta_1\overline{Y}, \overline{e''} = \theta\theta_2\overline{e'}\end{array}} \quad \text{(FIINV)}$$

$$\frac{\Gamma(x) = \overline{U} \to U \qquad \Gamma \vdash_{\theta_0} \overline{e} : \overline{S} \Rightarrow \overline{e'} \qquad \theta_0\Gamma \vdash_\theta \theta_0\overline{S} <: \theta_0\overline{U}}{\begin{array}{c}\Gamma \vdash_{\theta\circ\theta_0} x(\overline{e}) : U \Rightarrow x(\overline{e''}) \\ \text{where } \overline{e''} = \theta\overline{e'}\end{array}} \quad \text{(FIAPPFUNC)}$$

$$\frac{\Gamma(x) = U \equiv T \qquad \Gamma \vdash_{\theta_0} e : S \Rightarrow e' \qquad \theta_0\Gamma \vdash_\theta \theta_0 S <: \theta_0 U}{\begin{array}{c}\Gamma \vdash_{\theta\circ\theta_0} x \triangleright (e) : T \Rightarrow e \triangleright (e'') \\ \text{where } e'' = \theta e'\end{array}} \quad \text{(FIAPPEQ)}$$

Figure 11. Type Inference

ference does not interfere with method dispatching, and does not directly incorporate any OO concepts.

Logic Programming The SLD resolution is what Prolog [31] is based on. In Prolog, a proof that a query is satisfiable is usually less relevant, while in Eria, the argument for the implicit parameter, the counterpart of the proof, can be used as functions in the method body.

Type classes [12] in Haskell incorporate logic programming into a functional programming language, which is probably one of the reasons why Haskell is often used as the host language for EDSLs. Eria incorporates logic programming into an OO programming language. The rules used in Haskell are designed to replace ad hoc polymorphism, hence are more complex and not based on SLD resolution. As a result, sometimes type markers are needed in EDSLs.

Dependent Types There are functional programming languages that support explicitly the notion of proofs [8] (and references therein) through dependent types. A dependently typed language usually either has a type language that is separate to the term language, or has only one language for both its types and its terms. The expressibility of languages that support dependent types lies in the power of the dependent type theory [21]. A language that supports dependent types is usually capable of statically guaranteeing various properties such as length preservation and permutation in a sort algorithm and has the potential of being practical host languages for EDSLs, if it has a strong type inference algorithm.

EDSL Most of the research on EDSLs is in the functional setting. There are proposals for embedding typed languages into typed host languages, based on GADT [16, 36], dependent types [8, 28, 37], ordinary functions [4], or higher order abstract syntax [29]. Research is usually focused on creating tagless interpreters, which avoid tags, an overhead in EDSL implementations. MCS encoding does not have tags, but there might be other overhead such as object creation. There are also proposed ideas that can be used for embedding names, based on modal logic [6, 24, 25], or environment classifiers [32]. The design of labels is based on lightweight dependent type [1]. It can be viewed as syntax sugar for light weight dependent types that guarantees uniqueness through label type variables. Type equality is trivially equivalent to the equality of the desugared form in the nondependent type system. Similar concepts are also studied in extensible records [9, 20], but none of them are directly applicable in EDSLs. Implementing EDSLs in OO programming languages are less popular [13, 38].

Meta Languages Many programming languages support defining EDSLs to some extent. However, designing EDSLs in these languages usually needs to deal with many restrictions of the host language. On the other hand, many DSL systems provide meta languages for the specific domain of DSL definition [22, 34]. In these tools, the meta languages are themselves domain specific languages. Pattern languages, such as OMeta [35] and π [18], allows defining patterns and their semantics and can be also used as meta languages for EDSLs.

7. Discussion and Future Work

Library Code Generation The EriLex code generator supports generating library code that encodes the syntax and semantics of EDSLs from EDSL specifications written in the EriLex Specification Language (ESL). [39] Although the target language of EriLex is Java, it can be easily modified to generate code for Eria.

Usability In this paper, we demonstrated how Eria can be used to construct usable static language embeddings. There are other aspects of usability that needs to be studied further. Part of future work is to study how the current design works in EDSLs that are more complex than the STLC and formalize other parts of usability.

Proof We did not formally prove that our examples are usable embeddings, although it is clear that these examples are usable embeddings. However, for more complex languages or a general embedding framework [38], proving that the embedding is a uasable embedding is necessary. Also, given FE, we can prove that a general language embedding framework preserves the semantics of object language specifications.

Runtime Efficiency Improving the runtime efficiency of EDSLs in Eria is part of future work. We discuss a few options to improve the efficiency. The first option is to introduce the `final` keyword that functions in a similar way to Java. If a method is declared as final, than we can inline the method and perform further reductions. For example, we can eliminate unnecessary creation of intermediate objects. The second option is to construct an AST and partially evaluate

it. The third option is to use a well-typed code generator such as that in Example 4.2 and compile the generated code.

References

[1] Haskell wiki. http://www.haskell.org/haskellwiki/Dependent_type.

[2] jMock. http://www.jmock.org.

[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] Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. *SIGPLAN Not.*, 34(10):238–255, 1999.

[6] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270, New York, NY, USA, 1996. ACM.

[7] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 186–211, London, UK, 1998. Springer-Verlag.

[8] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121, New York, NY, USA, 2007. ACM.

[9] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical report, 1996.

[10] Max Rydahl Andersen Emmanuel Bernard Gavin King, Christian Bauer and Steve Ebersole. Hibernate reference documentation 3.3.2.ga, 2009.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[12] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[13] 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.

[14] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[16] Simon Peyton Jones. Simple unification-based type inference for gadts. pages 50–61. ACM Press, 2006.

[17] 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.

[18] Roman Knöll and Mira Mezini. π: a pattern language. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 503–522, New York, NY, USA, 2009. ACM.

[19] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[20] Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, December 2004.

[21] Per Martin-Lof. Intuitionistic type theory. *Bibliopolis*, 1984.

[22] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[23] Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for java. *ACM Trans. Program. Lang. Syst.*, 31(2):1–54, 2009.

[24] Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *J. Funct. Program.*, 15(6):893–939, 2005.

[25] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):1–49, 2008.

[26] Martin Odersky. The scala language specification version 2.7 draft, 2009.

[27] Doug Orleans. Incremental programming with extensible decisions. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 56–64, New York, NY, USA, 2002. ACM.

[28] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Not.*, 37(9):218–229, 2002.

[29] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM.

[30] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[31] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[32] Walid Taha and Michael Florentin Nielsen. Environment classifiers. *SIGPLAN Not.*, 38(1):26–37, 2003.

[33] Aaron Mark Ucko. Predicate dispatching in the common lisp object system, 2001.

[34] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[35] Alessandro Warth and Ian Piumarta. Ometa: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM.

[36] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, 2003.

[37] Hongwei Xi and Dana Scott. Dependent types in practical programming. In *In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1998.

[38] Hao Xu. A general framework for method chaining style embedding of domain specific languages. *UNC Technical Report*, 2009.

[39] Hao Xu. Erilex: An embedded domain specific language generator (to appear). In *TOOLS 2010*, 2010.