A General Framework for Embedding Domain Specific Languages into Object-Oriented Programming Languages

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

William Miao
School of Information and Library Science
University of North Carolina at Chapel Hill
wemiao@email.unc.edu

Abstract

In this paper, we introduce a general framework of language embedding in OO host languages. Unlike other approaches, which embed object language programs as nested calls to functions or data constructors, our framework embeds object language programs as a chain of method invocations. Our framework allows embedding a wider range of object language grammars into an OO host language, while still allowing object language programs to be type checked by the host language compiler and to run efficiently at run time. We demonstrate our framework using ELIA, our extension to Java with automatically constructed parameters (ACPs), and show how we can encode the parser, AST builder, type checker, and interpreter for the simply typed lambda calculus (STLC) in a concrete syntax without redundant brackets or commas. Our framework is only based on the basic OO features, generics, and ACPs, and do not require higher-order functions or generalized abstract data types (GADTS) in the host language. It is a generalization of our previous framework [34], which can be fully implemented in Java without ACPs.

Keywords Embedded Domain Specific Language, Object-Oriented Programming, Automatically Constructed Parameter, Term Inference, Language Extension, General Framework

1. Introduction

One popular way of implementing a new language is "embedding" the language in an existing language [18]. In this application domain, the former is called the object language and the latter is called the host language. The "embedding" approach allows the infrastructure of the host language, such as its compiler, to be reused by the object language, and the properties of the host language, such as type safety, to carry over to the object language. This approach is especially appealing in implementing domain specific languages (DSLs). It enables creating new DSLs with state of the art support infrastructures, such as secure and scalable virtual machines and developer tools, that are generally available only to mainstream programming languages.

The main advantage of the "embedding" approach, compared to other approaches to implementing embedded DSLs, is that it requires minimum coupling with the host language. It does not require meta language constructs in the host language or extensibility of the host language compiler. The embedded DSLs rely on the syntax and semantics of the host language, but not how the syntax and semantics are defined or implemented. As a result, the "embedding" approach allows the host languages to evolve without breaking the embedded DSLs, as long as the host languages are backward compatible in syntax and semantics. Another advantage of this approach is that no external tools or preprocessors are needed for compiling embedded DSL code, as embedded DSL code is host language code.

Despite its advantages, the gap between research and industry prevents embedding from being widely adopted. One problem is that programming paradigms differ. On the one hand, most research that produced methodologies for embedding one typed language into another was done in a functional setting; on the other hand, most widely used and well-supported programming languages today, such as the Java programming language, are object-oriented (OO) programming languages. These languages usually do not directly support many key functional features, such as higher-order functions and generalized algebraic data types (GADTs). Another problem is that most research focuses on building tagless interpreters that help achieve better performance at runtime. Most of the existing methodologies for typed embedding require that embedded DSL programs be written in a restricted grammar.

To bridge this gap, we need to find a method that allows embedding a wider range of object language grammars into OO host languages, while still allowing object language programs to be type checked by the host language compiler and to run efficiently at run time. The key idea is to simulate the object language parser using the host language type checker, rather than using the host language parser as in other approaches [4, 32]. We briefly explain our idea using terminologies of pushdown automata. Given a grammar of an object language, a well-formed object language program is a string of input symbols that is accepted by a corresponding pushdown automaton. We encode this automaton using classes and methods in an OO host language. More specifically, we map

- an input symbol to a method invocation in the host language; and
- a string of input symbols to a chain of method invocations (COMICs) in the host language, where the return value of one method invocation becomes the object on which the next method invocation is performed. This style of embedding is sometimes called the method chaining style.

Then, we assign host language types to the methods so that type checking COMICs simulates the automaton. This way, only well-formed object language programs can be mapped to well-typed host language COMICs.

Unlike traditional approaches, the COMIC embedding

- decouples the concrete syntax of the object language from representations of abstract syntax trees (ASTs) in the host language, thereby allowing more flexibility in the object langauge;
- provides better hints to the host language semantic editor, thereby enabling "parsing/type checking/AST building as you go" and more accurate reuse of host language editor functionalities, such as autocompletion, in the object language.

In practice, various DSLs in tools suchs as jMock [2], Hibernate Criteria Query [16], and Google Protocol Buffers [1] are implemented in the method chaining style.

In this paper, we introduce a general framework of language embedding in OO host languages. In addition to the features of the general COMIC embedding, this framework

- allows defining AST builders, type checkers, and interpreters separately;
- works with existing methodologies for defining tagless interpreters.

We demonstrate our general framework using ELIA, our extension to Java. A previous framework [34] by the first author, which can be fully implemented in Java, is a special case of this framework. We demonstrate our approach by encoding a parser, AST builder, type checker, and interpreter for the STLC with a concrete syntax in ELIA.

2. ELIA

In this section we informally introduce ELIA, our extension of Java.

2.1 Overview of ELIA

The main motivation for introducing ELIA is to encode complex automata that cannot be fully encoded in the type system of Java. In designing ELIA, we created the following criteria: it should be a conservative extension, require minimal change in the host language, have minimal impact on the runtime performance, not be useful only for the task of language embedding, and not compromise type safety. ELIA satisfies all the above criteria.

The basic idea of ELIA is combining techniques of logical programming with those of type theory. ELIA allows programmers to define a list of logical premises and put logical queries in their programs. The compiler then performs these logical queries at compile time. For a successful query, the compiler also constructs a "proof", which can be used at runtime as a computational term. The results of the compilation, such as whether the compilation succeeds, are determined by the results of the logical queries. The logical premises and logical queries are similar to clauses and queries in Prolog [28].

2.2 Automatically Constructed Parameters

The most important concept in ELIA is *automatically constructed parameters* (ACPs). An ELIA method definition can declare a list of ACPs in addition to regular Java method parameters. The types of ACPs correspond to logical queries.

There are two main differences between an ACP and a regular Java method parameter. First, an ACP can have a function type as well as a regular Java type. When an ACP has a function type, it can be used as a function in the method definition. For example, the following method definition declares an ACP `p` of type `S->String`[1] besides a regular Java method parameter `x` of type `S`, where highlighted text is ELIA-specific syntax. ELIA uses the vertical line to delimit ACPs from regular Java method parameters.

```
static <S> String deepDeref(S x |S->String p ) {
    return p (x);
}
```

In general, the syntax of an ACP list is

$\mathtt{T}_1\ \mathtt{p}_1,\ldots,\mathtt{T}_m\ \mathtt{p}_m$

where $m \in \mathbb{N}$, $\mathtt{T}_i$ is a function type of the form $(\mathtt{P}_1,\ldots,\mathtt{P}_{n_i})\texttt{->}\mathtt{R}_i$, where $n_i \in \mathbb{N}$, and $\mathtt{p}_i$ is an ACP name, for all $i \in \{1,\ldots,m\}$. A more complex example is

```
static double gravity(
    double m1, double m2, double r,
    |double G, (double,double)->double prod ) {
    return G*m1*m2/prod(r,r);
}
```

We omit the parentheses in an ACP type when the function type has only one parameter. In the following discussion, we write $(\mathtt{P}_1,\ldots,\mathtt{P}_n)\texttt{->}\mathtt{R}$ in a more readable form $(\mathtt{P}_1,\ldots,\mathtt{P}_n)\rightarrow\mathtt{R}$, and refer to $(\mathtt{P}_1,\ldots,\mathtt{P}_{n_i})$ as the parameter type of the ACP.

Second, in a method invocation, the programmer does not specify the values of ACPs, but lets the ELIA compiler automatically construct them according to their types. For example,[2]

```
deepDeref(new Reference<String>("xyz"))
```

Before explaining how ACPs work, we need to introduce another concept. In ELIA, programmers can define building blocks of ACP values called *ACP constructors*. An ACP value constructor is similar to a static Java method, but differs from a static Java method in two aspects:

1. It is defined with an ELIA keyword `implicit`.
2. All of tts parameters are ACPs.

For example, we can define the following ACP constructor:

```
implicit <T> T deref(Reference<T> p) {
    return p.get();
}
```

This ACP constructor provides a building block for building ACP values.

In general, we denote the type of every ACP constructor `acpc`

$\texttt{implicit <}\mathtt{T}_1,\ldots,\mathtt{T}_n\texttt{> R acpc(}\mathtt{P}_1\ \mathtt{x}_1,\ldots,\mathtt{P}_m\ \mathtt{x}_m\texttt{) \{ ... \}}$

by

$\texttt{<}\mathtt{T}_1,\ldots,\mathtt{T}_n\texttt{>}(\mathtt{P}_1,\ldots,\mathtt{P}_m)\rightarrow\mathtt{R}$

where $m, n \in \mathbb{N}$, `R` is the return type, $\mathtt{P}_1,\ldots,\mathtt{P}_m$ are parameter types, and $\mathtt{T}_1,\ldots,\mathtt{T}_n$ are type parameters. We may omit the parentheses in the type if the parameter list has exactly one parameter. For example, the type of the ACP constructor we just defined is $\texttt{<T>Reference<T>}\rightarrow\mathtt{T}$. The types of ACP constructors correspond to logical premises. They provide important heuristics to the ELIA compiler when it tries to construct values for ACPs.

Intuitively, an ACP type $(\mathtt{P}_1,\ldots,\mathtt{P}_m)\rightarrow\mathtt{R}$ asks the question: "can you construct an ACP value which is a function that maps any values of types $\mathtt{P}_1,\ldots,\mathtt{P}_m$ to a value of type `R`?" The algorithm

[1] `S->String` is a function type. The function takes in a value of type `S` and returns a value of type `String`.

[2] A `Reference<T>` object stores a reference to another object of type `T`. It is defined in the standard Java package `java.lang.ref`. The `Reference<T>` type has a method named `get` which returns the object referred to by the reference object.

used by the ELIA compiler to construct values for ACPs is similar to “backward chaining” in automatic theorem proving, which we demonstrate using an example.

Example 2.1. In this example, we use the method `deepDeref` and the ACP constructor `deref` we just defined. Suppose that we have defined the following local variables:

```
String s = "xyz";
Reference<String> rs = new Reference<String>(s);
Reference<Reference<String>> rrs =
    new Reference<Reference<String>>(rs);
```

In order to retrieve the string object from the three variables, we need to dereference them zero, one, and two times, respectively. If this operation is implemented in Java, then we need to either write three separate methods, or to write one method that finds out the types using reflection dynamically at runtime. In ELIA, we can write one method, in a statically type-safe fashion. The method is the `deepDeref` method that we defined earlier.

Now, let us take a look at what happens when ELIA compiles the following method invocations:

```
deepDeref(s)
deepDeref(rs)
deepDeref(rrs)
```

Recall that the type of the ACP `p` of the `deepDeref` method is `S->String`, where `S` is the type of the regular Java parameter `x`. Since the arguments to the regular parameters have different types, the ACPs also have different types. We show the ACP types for the three method invocations in Table 1. The constructed ACP values are shown next to their types, and we denote function composition by $\circ$.

For the first method invocation, the ACP type asks the question “can you construct a function that maps a value of type `String` to a value of type `String`?” The ELIA compiler assigns the ACP the identity function `id`. We can write this process in the style of natural deduction [12, 14]. Here, we abbreviate `String` to `S`.

$$\dfrac{\dfrac{}{\texttt{S}} \quad \dfrac{}{\texttt{S}\to\texttt{S}}\,\texttt{id}}{\texttt{S}}\to E$$

For the second method invocation, the ACP type asks the question “can you construct a function that maps a value of type `Reference<String>` to a value of type `String`?” The ELIA compiler starts by trying to find a function that returns a value of type `String`. It goes through all ACP constructors, asking the question “could this ACP constructor return a value of type `String`?” It finds the ACP constructor `deref`, which has type `<T>Reference<T>`$\to$`T`, and instantiates the type variable `T` to type `String` by unification. As the parameter type of this instance of `deref` is the same as the parameter type of the ACP, the compiler assigns the ACP the function `deref`. We can write this process in the style of natural deduction. Here, we abbreviate `Reference` to `R`.

$$\dfrac{\dfrac{}{\texttt{R<S>}} \quad \dfrac{}{\texttt{R<S>}\to\texttt{S}}\,\texttt{deref}}{\texttt{S}}\to E$$

For the third method invocation, the ACP type asks the question “can you construct a function that maps a value of type `Reference<Reference<String>>` to a value of type `String`?” The ELIA compiler starts by trying to find a function that returns a value of type `String`. It goes through all ACP contructors, asking the question “could this ACP constructor return a value of type `String`?” It finds the ACP constructor `deref`, which has type `<T>Reference<T>`$\to$`T`, and instantiates the type variable `T` to type `String` by unification. But in this case, the parameter type of this instance of `deref` is `Reference<String>`, which is different from the parameter type `Reference<Reference<String>>` of the ACP. The compiler proceeds by trying to find a function that returns a value of type `Reference<String>`. It finds `deref` again, but initiates the type variable `T` to type `Reference<String>`. This time, this instance of `deref` takes in an argument of type `Reference<Reference<String>>`, which is exactly what we are looking for. Thus, the ELIA compiler constructs a function by composing two `deref`s together. We can write this process in the style of natural deduction.

$$\dfrac{\dfrac{\dfrac{}{\texttt{R<R<S>>}} \quad \dfrac{}{\texttt{R<R<S>>}\to\texttt{R<S>}}\,\texttt{deref}}{\texttt{R<S>}}\to E \qquad \dfrac{}{\texttt{R<S>}\to\texttt{S}}\,\texttt{deref}}{\texttt{S}}\to E$$

The general process of constructing ACP values is described below:

- First, the compiler converts types of all (visible) ACP constructors into logical expressions, which are used as premises in the inference.
- Then, the compiler converts the types of ACPs into logical expressions, which are used as queries in the inference.
- Given the premises and queries, the ELIA compiler uses backward chaining to find a proof of the queries based on the premises.
- Finally, the proof is converted back to an ACP value.

2.3 Equality Type

In addition to function types, ACPs can also have equality types and inequality types. An equality type has the form $\texttt{A}_1\texttt{==}\texttt{A}_2$, where $\texttt{A}_1$ and $\texttt{A}_2$ are regular Java types. For example,

```
static <S,R> R eq(S x|S==R p) {
    return p(x);
}
```

An ACP with equality type $\texttt{A}_1\texttt{==}\texttt{A}_2$ can be used as a function of type $\texttt{A}_1\to\texttt{A}_2$, but its constructed value can only be the identity function `id`. The ACP enforces the definitional type equality $\texttt{A}_1=\texttt{A}_2$.

Similarly, ELIA also supports the inequality type. An inequality type has the form $\texttt{A}_1\texttt{!=}\texttt{A}_2$, where $\texttt{A}_1$ and $\texttt{A}_2$ are regular Java types. For example,

```
static <S,R> S ineq(S x|S!=R q) {
    return x;
}
```

Unlike other ACPs, an ACP with an inequality type cannot be used as a function.

2.4 Cut and Customizable Error Messages

ELIA supports a special syntax `!` which, when put into an ACP list, works similarly to the “cut” predicate in Prolog.[3] For example, given the following definitions:

```
interface A { ... }
interface B { ... }
implicit A r1() { ... }
implicit B r2() { ... }
```

the method

```
static <V> B ng(|()->V p,!,V==B q) {
    return q(p());
}
```

can never be invoked. The reason is ELIA matches ACPs with ACP constructors in the order the ACP constructors are defined in the

[3] By default, all ACP values are constructed locally in a method invocation. Therefore, there is no need for “cut” between method invocations.

method call	S	ACP type	constructed value
`deepDeref(s)`	`String`	`String->String`	`id`
`deepDeref(rs)`	`Reference<String>`	`Reference<String>->String`	`deref`
`deepDeref(rrs)`	`Reference<Reference<String>>`	`Reference<Reference<String>>->String`	`deref`$\circ$`deref`

Table 1. ACP Types and Constructed ACP Values

program. that the ACP `p` matches `r1` first, which makes `V=A` and fails the second ACP `q`. `!` prevent the compiler from backtracking and matching the ACP `p` with `r2`.

Another important usage of "cut" is that it can act as a point where programmers can define customized error messages. Customizable error messages can be used to bridge the gap in error reporting between a host language and an embedded language. In ELIA, programmers can define customized error messages after `!` using the `error` keyword. ELIA types can be concatenated with strings to form error messages. We demonstrated its syntax using the following example:

```
class A {
  <T> T is(S n|
    ! error (S+" is not convertible to "+T),
    S->T p) {
    ...
  }
}
```

2.5 Type Inference

In addition to constructing ACP values, the ELIA compiler can also infer the instantiation of free type variables when they are in an ACP type. For example, we can define a more general version of the function `deepDeref` we defined earlier.

```
static <S,R,T> R deepDeref2(S x|S->R p,
    S!=Reference<T> q) {
    return p(x);
}
```

In this method, the ACP has type `S->R`.

In a method invocation such as

```
deepDeref2(rs)
```

where `rs` is a local variable of type `Reference<String>`, the compiler unifies the ACP type `Reference<String>->R` with `deref`'s type `Reference<T>->T`, and infers that `R=String`.

3. Our General COMIC Embedding Framework

In this section we introduce our general COMIC embedding framework. First, we introduce the *algebraic multistack pushdown automaton* (ampda), our variant of pushdown automata. We use amdpas as intermediate encoding devices for parsers, AST builders, and type checkers, so that they can be defined separately. Then, we introduce our general encoding framework, which is based on the idea of representing ampdas in the type system of ELIA. Finally, we show an example of encoding a parser, an AST builder, a type checker, and an interpreter for the Simply Typed Lambda Calculus (STLC) in ELIA using our framework. Using this encoding, we can embed STLC programs as shown in the following example.

Example 3.1. The STLC program **fun** $x \Rightarrow$ **fun** $y \Rightarrow y\ x$ can be encoded by the following COMIC:

```
prog.fun().x().fun().y().arrow().y().x().run()
```

Note that we encode the grammar rule "**fun** binds the longest subexpression possible," which is similar to other rules such as "'else' associates with the closes 'if.'"

3.1 Preliminaries

Our general methodology is to model the parser as a variant of pushdown automata (pda). The main reason that we choose this variant of pda is that it is easier to present in this paper and more straightforward to encode in ELIA than the standard pda.

Definition 3.2. A (one-sorted algebraic) *signature* is a pair (Σ, ar) where

- Σ is a finite set of function symbols
- ar is a function from Σ to $\mathbb{N}$ which assigns an arity to every function symbol.

We write a signature with $\Sigma = \{f_1, \ldots, f_n\}$ and $ar(f_1) = d_1, \ldots, ar(f_n) = d_n$ as $\{f_1 : d_1, \ldots, f_n : d_n\}$. The purpose of introducing signatures is so that we can use terms generated from signatures as stack elements in our automaton defined below. Following the notation of "Foundations for Programming Languages" [22], we denote the terms over a signature $\mathcal{S}$ by $\mathit{Terms}(\mathcal{S})$, and the terms over a signature and a finite set V of variable symbols by $\mathit{Terms}(\mathcal{S}, V)$.

Definition 3.3. An *algebraic multistack pushdown automaton (ampda)* is a tuple $(Q, \Sigma, \mathcal{S}_1, \ldots, \mathcal{S}_n, \delta, z, F)$, where

- $n \geq 1$ is the number of stacks
- Q is a finite set of states
- Σ is a finite set of input symbols
- $\mathcal{S}_i$ is a signature that generates the set of terms that can be pushed onto the ith stack, for $i \in \{1, \ldots, n\}$
- δ is a partial function from $Q \times \mathit{Terms}(\mathcal{S}_1)^* \times \ldots \times \mathit{Terms}(\mathcal{S}_n)^* \times \Sigma$ to $Q \times \mathit{Terms}(\mathcal{S}_1)^* \times \ldots \times \mathit{Terms}(\mathcal{S}_n)^*$
- $z \in Q \times \mathit{Terms}(\mathcal{S}_1)^* \times \ldots \times \mathit{Terms}(\mathcal{S}_n)^*$ is the initial configuration
- $F \subset Q \times \mathit{Terms}(\mathcal{S}_1)^* \times \ldots \times \mathit{Terms}(\mathcal{S}_n)^*$ is the finite set of accepting configurations

A configuration is an element in $Q \times \mathit{Terms}(S_1)^* \times \ldots \times \mathit{Terms}(S_n)^*$. We denote a configuration by $q\overline{s}$, where $\overline{s}$ is a shorthand for $(s_1, \ldots, s_n)$, the n-tuple of stacks. We assume that in every stack there is a special element $\perp$ which denotes *bottom of stack*. We write a stack consisting terms $A_1, \ldots, A_n$, where A_n is the top of the stack, as $\perp A_1 \ldots A_n$. We may write a configuration $q\overline{s}$ in the vector form

$$q \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}$$

To improve the readability of ampda definitions, we represent the partial function δ of an ampda as a transition table. In a transition table, each row corresponds to a state and each column corresponds to an input symbol. The first row indicates the input symbols and the first column indicates the states. We use a pair (q, a), where q is a state and a is an input symbol, as the coordinates of a cell in a transition table. In each cell (q, a), we write the transition rules that are activated by state q and input symbol a.

Given a finite set V of variable symbols, we can write two kinds of rules in a cell.

The first kind of rules are transition rules. A transition rule represents a transition from a configuration $q\overline{s}$ to another configuration $q'\overline{t}$, where $s_i, t_i \in Terms(S_i, V)^*$ for all $i \in \{1, \ldots, n\}$. Intuitively, a transition rule $\overline{s}/q'\overline{t}$ in cell (q, a) means:

If the current configuration matches $q\overline{s}$, then transition to $q'\overline{t}$.

A transition rule is represented by the vector form

$$\begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} /q' \begin{bmatrix} t_1 \\ \vdots \\ t_n \end{bmatrix}$$

We omit the brackets if there is only one stack. We allow multiple transition rules in a cell as long as there is only one rule that is applicable at a time. If there are multiple rules in a cell, we can further divide the rules into two groups. The first group of rules are "stop rules." After applying a "stop rule," the transition in the cell is complete and the ampda proceeds to process the next input symbol, if any. The second group of rules are "continuation rules." After a continuation rule is applied, the ampda continues to apply rules in the same cell until it hits a "stop rule." Every cell that is not empty must contain at least one "stop rule."[4] We write a $\triangleleft$ at the end of a "stop rule" and an $*$ at the end of a "continuation rule."

The second kind of rules are lookup rules. A lookup rule instantiates variables in the rule by performing recursive lookup on a stack. It has the form $st_i =_{v_1 \ldots v_n} A \ldots$, where $n \in \mathbb{N}$, $v_1 \ldots v_n$ is a list of variable symbols that appear in A, st_i indicates that the lookup is performed on the ith stack, and $A \in Terms(S_i, V)$ is a term pattern that the rule looks for on the stack. The rule starts from the top of the stack and goes down until it finds a term A' such that A and A' are unifiable. If it reaches bottom of stack, then it fails.

In this paper, we consider only ampdas that can be represented by a transition table with a finite number of rules in every cell.

Example 3.4. The following is an example transition table for an ampda that accepts the langauge $\{a^n b^n | n \in \mathbb{N}\}$.

Initial configuration = $0\bot$

Accepting configurations = $\{0\bot, 1\bot\}$

	a	b
0	$/0B$	$B/1$
1		$B/1$

Example 3.5. The following is an example transition table for an ampda that accepts the langauge $\{a^n b^n | n \in \mathbb{N}\} \cup \{a^n c | n \in \mathbb{N}\}$. Here, we need multiple rules in a cell of the transition table (which simulates the behavior of a jump pda [3]).

Initial configuration = $0\bot$

Accepting configurations = $\{0\bot, 1\bot\}$

	a	b	c
0	$/0B$	$B/1$	(1) $B/0$ $*$ (2) $\bot/1\bot$ $\triangleleft$
1		$B/1$	

3.2 Encoding Ampdas In ELIA

The general idea of encoding an ampda in ELIA is based on the following interpretation of the transition table.

Representing Terms Generated by a Signature We define a ELIA type constructor T_f of arity n for each function symbol of arity n, and a type variable T_v for every variable symbol v. The general encoding function $en_0[-]$[5] from terms to ELIA types is defined as

$$\begin{aligned} en_0[v] &= T_v \\ en_0[f(A_1, \ldots, A_n)] &= T_f{<}en_0[A_1], \ldots, en_0[A_n]{>} \end{aligned}$$

where v is a variable symbol, f is a function symbol of arity n, and $A_1, \ldots, A_n$ are terms.

Representing Stacks of Terms The idea follows that of algebraic data types. We define two ELIA type constructors: $push$ of arity 2 and $T_\bot$ of arity 0. A stack of terms $\bot A_1 \ldots A_n$ is represented by type

$$push{<}en_0[A_n], push{<}en_0[A_{n-1}], \ldots, push{<}en_0[A_1], T_\bot{>}{>}{>}$$

In the transition rules, we usually only need to inspect the top portion of a stack. The top portion of a stack consisting of terms $A_1, \ldots, A_n$ is represented by type

$$push{<}en_0[A_n], push{<}en_0[A_{n-1}], \ldots, push{<}en_0[A_1], v{>}{>}{>}$$

where v is a type variable that does not appear in $en_0[A_1], \ldots, en_0[A_n]$. The general encoding function $en_1[-]$ from stacks of terms to ELIA types is defined as

$$\begin{aligned} en_1[\varepsilon] &= v \\ en_1[\bot] &= T_\bot \\ en_1[sA] &= push < en_0[A], en_1[s] > \end{aligned}$$

where ε is the empty sequence, s is a possibly empty sequence of stack terms, A is a stack term, and v is a fresh type variable. For example, given a signature $\{B : 0\}$, the encoding of a stack $\bot BBB$ looks like $push{<}T_B, push{<}T_B, push{<}T_B, T_\bot{>}{>}{>}$, and the encoding of the top portion BBB of a stack looks like $push{<}T_B, push{<}T_B, push{<}T_B, v{>}{>}{>}$.

Representing States, Input Symbols, and Configurations Given an ampda with n stacks, we define an ELIA type constructor C_q of arity n for each state q, and a method M_a in C_q for each state and input symbol pair (q, a) if and only if the cell (q, a) is not empty. For example, the transition table in Example 3.5 yields the ELIA type constructors and methods shown in the following table:

ELIA type	methods
C_0	M_a, M_b, M_c
C_1	M_b

Given the representation of states and input symbols, the general encoding function $en_2[-]$ from configurations to ELIA types is defined as

$$en_2[q\overline{s}] \quad = \quad C_q{<}en_1[s_1], \ldots, en_1[s_n]{>}$$

where $q\overline{s}$ is a configuration.

Representing Transition Rules We defined ELIA types T_* and $T_\triangleleft$ indicating "continuation rules" and "stop rules," respectively. For every cell (q, a) that is not empty, we define an ELIA type constructor $G_{q,a}$ of arity 2. For each transition rule $\overline{s}/q'\overline{t}$ in cell (q, a), we defined an ACP constructor of type (to make it easier to read, we assume that free type variables are implicitly universally quantified)

$$G_{q,a}{<}en_2[q'\overline{t}], \tau_1{>} \rightarrow G_{q,a}{<}en_2[q\overline{s}], \tau_2{>}$$

[4] The main purpose for introducing these two groups of rules is so that we can encode a wider range of transition functions in a transition table. No transition function in any ampda should diverge. If application of the rules diverges, then the rules are not valid represention of a transition function.

[5] We use square brackets for metalevel functions to avoid confusion with parentheses which are used to represent subterms.

where type τ_1 and τ_2 are either T_* or $T_\triangleleft$. [6] The left hand side represents the target configuration and the right hand side represents the source configuration. The ELIA ACP constructor code looks like

```
implicit <...>
G_{q,a}<en_2[qs̄],τ_2> r(G_{q,a}<en_2[q't̄],τ_1> p) { ... }
```

For every method M_a in ELIA type $C_q{<}S_1, \ldots, S_n{>}$, where $S_1, \ldots, S_n$ are ELIA types, we set its return type to ϕ, where ϕ is a fresh type variable, and define an ACP of type

$$G_{q,a}{<}\phi, T_\triangleleft{>} \to G_{q,a}{<}C_q{<}S_1, \ldots, S_n{>}, T_*{>}$$

Intuitively, the right hand side represents the current configuration and the left hand side represents that we want to apply rules in cell (q, a) repeatedly until it hits a "stop rule."

The ELIA code looks like

```
interface C_q<S_1,...,S_n> {
  <...> φ M_a(
  | G_{q,a}<φ,T_◁>->G_{q,a}<C_q<S_1,...,S_n>,T_*> p);
  ...
}
```

The transition rules in cell (q, a) are simulated by the ACP constructors and the ACP as follows. Suppose that we have a configuration $q\overline{s}$. Its representation in ELIA is type $en_2[q\overline{s}]$. Every method M_a in this ELIA type has an ACP of type $G_{q,a}{<}\phi, T_\triangleleft{>} \to G_{q,a}{<}en_2[q\overline{s}], T_*{>}$. To construct a value for this ACP, the compiler starts by looking for an ACP constructor instance whose return type is $G_{q,a}{<}en_2[q\overline{s}], T_*{>}$. If it finds an ACP constructor instance of type $G_{q,a}{<}en_2[q'\overline{t}], T_*{>} \to G_{q,a}{<}en_2[q\overline{s}], T_*{>}$, then it proceeds by looking for another ACP constructor instance whose return type is $G_{q,a}{<}en_2[q'\overline{t}], T_*{>}$, which simulates a "continuation rule" transition from $q\overline{s}$ to $q'\overline{t}$; if it finds an ACP constructor instance of type $G_{q,a}{<}en_2[q'\overline{t}], T_\triangleleft{>} \to G_{q,a}{<}en_2[q\overline{s}], T_*{>}$, then it stops and assigns type variable ϕ the type $en_2[q'\overline{t}]$, which simulates a "stop rule" transition from $q\overline{s}$ to $q'\overline{t}$. The two cases can be written in the style of natural deduction as follows: (we omit all labels in the deductions)

$$* : \quad \frac{\dfrac{\vdots}{G_{q,a}{<}en_2[q'\overline{t}],T_*{>}} \qquad \dfrac{}{G_{q,a}{<}en_2[q'\overline{t}],T_*{>}\to G_{q,a}{<}en_2[q\overline{s}],T_*{>}}}{G_{q,a}{<}en_2[q\overline{s}],T_*{>}}$$

$$\triangleleft : \quad \frac{\dfrac{}{G_{q,a}{<}en_2[q'\overline{t}],T_\triangleleft{>}} \qquad \dfrac{}{G_{q,a}{<}en_2[q'\overline{t}],T_\triangleleft{>}\to G_{q,a}{<}en_2[q\overline{s}],T_*{>}}}{G_{q,a}{<}en_2[q\overline{s}],T_*{>}}$$

Next, we show an example. The following transition rules from cell $(0, c)$ in Example 3.5

(1)	$B/0$	$*$
(2)	$\bot/1\bot$	$\triangleleft$

can be represented by ACP constructors of the following types:

$$<v>G_{0,c}{<}C_0{<}v{>}, T_*{>} \to G_{0,c}{<}C_0{<}push{<}T_B, v{>}{>}, T_*{>}$$
$$G_{0,c}{<}C_1{<}T_\bot{>}, T_\triangleleft{>} \to G_{0,c}{<}C_0{<}T_\bot{>}, T_*{>}$$

Representing Lookup Rules We define an ELIA type constructor L of arity 2 and two ACP constructors. The purpose of introducing L is so that we can use $L{<}v, \phi{>}$ to denote "look up ϕ in stack v." The first ACP constructor has type

$$() \to L{<}push{<}\phi, v{>}, \phi{>}$$

[6] We are not writing the type the other way around as

$$G_{q,a}{<}en_2[q\overline{s}], \tau_2{>} \to G_{q,a}{<}en_2[q'\overline{t}], \tau_1{>}$$

because of backchaining.

e	$\to$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	**fun** $x \Rightarrow e$	abstraction
	$\mid$	(e)	parentheses

Figure 1. Concrete Syntax: abstraction binds the longest subexpression possible

Initial Configuration = $0\bot$
Acception Configurations = $\{1\bot\}$

	x	**fun**	$\Rightarrow$	(	)
0	$/1$	$/2$		$/0P$	
1	$/1$	$/2$		$/0P$	$P/1$
2	$/3$				
3			$/0$		

Table 2. Parser Transition Table

and the second ACP constructor has type

$$(\chi ! = \phi, L{<}v, \phi{>}) \to L{<}push{<}\chi, v{>}, \phi{>}$$

where v, ϕ, and χ are type variables (free variables are implicitly universally quantified). The ELIA code looks like

```
implicit <...> L<push<φ,v>,φ> r0() { ... }
implicit <...>
L<push<χ,v>,φ> r1(χ!=φ q, L<v,φ> p) { ... }
```

Intuitively, they encode the following:

> If the top element of stack matches ϕ, success.
> Otherwise, pop top element $\chi$and try again.
> If it reaches bottom of stack, failure.

For each lookup rule $st_i =_{v_1 \ldots v_n} A \ldots$ in cell (q, a), we define an ACP of type $L{<}S_i, en_0[A]{>}$ in method M_a of type $C_q{<}S_1, \ldots, S_n{>}$ and extend the type constructor $G_{q,a}$ with type variables $en_0[v_0], \ldots, en_0[v_n]$. The ELIA code looks like

```
interface C_q<S_1,...,S_n> {
  <...> φ M_a(
  | L<S_i,en_0[A]> q,
    G_{q,a}<φ,T_◁ ,en_0[v_0],...,en_0[v_n] >
  ->G_{q,a}<C_q<S_1,...,S_n>,T_* ,en_0[v_0],...,en_0[v_n] > p);
  ...
}
```

We will show an example of lookup rules in Section 3.3.3.

3.3 Example: Embedding Simply-Type Lambda Calculus (STLC) with A Concrete Syntax

In this subsection, we show an example of embedding the STLC in ELIA using our general framework. We introduce the embedding in four parts: parser, AST builder, type checker, and interpreter. The syntax of the STLC is shown in Figure 1.

3.3.1 Parser

The parser ampda is shown in Table 2. We follow the steps in our general framework.

1. For terms, as there is only one stack and one function symbol P used in the stack, it suffices to define an ELIA type `P`.
2. For stacks, we define ELIA types `push` and `bot`, representing *push*, and $T_\bot$, respectively.

3. For states, input symbols, and configurations, we define ELIA types and methods as shown in the following table:

ELIA type	defined methods
`C0`	`x`, `fun`, `l`
`C1`	`x`, `fun`, `l`, `r`
`C2`	`x`
`C3`	`arrow`

where `C0`, `C1`, `C2`, and `C3` represent states 0, 1, 2, and 3, respectively, `x`, `fun`, `arrow`, `l`, and `r` represent input symbols x, **fun**, $\Rightarrow$, (, and), respectively.

4. For the transition rules, we define an ELIA type constructor representing $G_{q,a}$ in our framework for every non-empty cell (q, a), as shown in the following table:

	x	**fun**	$\Rightarrow$	(	)
0	`G0x`	`G0fun`		`G0l`	
1	`G1x`	`G1fun`		`G1l`	`G1r`
2	`G2x`				
3			`G3arrow`		

We define ELIA types `C` (Continuation) and `O` (Stop), which represents $*$ and $\triangleleft$, respectively.
For each rule, we define an ACP constructor. The type declaration part of the ACP constructor definitions are shown in the following table. We omit the key word `implicit`.

q, a	ACP constructor
$0, x$	`<V> G0x<C0<V>,C> r0(G0x<C1<V>,O> p)`
$1, x$	`<V> G1x<C1<V>,O> r1(G1x<C1<V>,C> p)`
$2, x$	`<V> G2x<C2<V>,C> r2(G2x<C3<V>,O> p)`
0, **fun**	`<V> G0fun<C0<V>,C> r3(G0fun<C2<V>,O> p)`
1, **fun**	`<V> G1fun<C1<V>,C> r4(G1fun<C2<V>,O> p)`
$3, \Rightarrow$	`<V> G4arrow<C4<V>,C> r5(G4arrow<C0<V>,O> p)`
0, (	`<V> G0l<C0<V>,C> r6(G0l<C0<push<P,V>>,O> p)`
1, (	`<V> G1l<C1<V>,C> r7(G1l<C20<push<P,V>>,O> p)`
1,)	`<V> G1r<C1<push<P,V>>,C> r8(G1r<C1<V>,O> p)`

We can write the full details of the methods from step 3.

```
interface C0<V> {
  <T> T x(|G0x<T,O>->G0x<C0<V>,C> p);
  <T> T fun(|G0fun<T,O>->G0fun<C0<V>,C> p);
  <T> T l(|G0l<T,O>->G0l<C0<V>,C> p);
}
interface C1<V> {
  <T> T x(|G1x<T,O>->G1x<C1<V>,C> p);
  <T> T fun(|G1fun<T,O>->G1fun<C1<V>,C> p);
  <T> T l(|G1l<T,O>->G1l<C1<V>,C> p);
  <T> T r(|G1r<T,O>->G1r<C1<V>,C> p);
}
interface C2<V> {
  <T> T x(|G2x<T,O>->G2x<C2<V>,C> p);
}
interface C3<V> {
  <T> T
  arrow(|G3arrow<T,O>->G3arrow<C2<V>,C> p);
}
```

5. We do not have lookup rules in this ampda, but we define an ELIA type constructor `L` without any method, which represents L in our framework. We will use it in the following subsubsections.

6. In fact, we can simplify some of the methods. If there is only one rule that applies in cell (q, a) and no element is popped from the stack, then we can determine the return type `T` and remove the ACP. For example, for method `x` in `C0`, we can determine the return type of the method by looking at the table above. The only ACP constructor that can be used to construct the ACP value is `<V> G0x<C0<V>,C> r0(G0x<C1<V>,O> p)`. Therefore, we can remove the ACP in the method declaration and replace the return type with `C1<V>`. The following is the simplified code:

```
interface C0<V> {
  C1<V> x();
  C2<V> fun();
  C0<push<P,V>> l();
}
interface C1<V> {
  C1<V> x();
  C2<V> fun();
  C0<push<P,V>> l();
  <T> T r(|G1r<T,O>->G1r<C1<V>,C> p);
}
interface C2<V> {
  C3<V> x();
}
interface C3<V> {
  C1<V> arrow();
}
```

3.3.2 AST Builder

We extend our parser with an AST builder. We introduce a new stack with signature $\mathcal{S} = \{x : 0, var : 1, app : 2, abs : 2, A : 1, A' : 1\}$. The general AST builder partial function $node[-]$ from sequences of input symbols to $Terms(\mathcal{S})$ that we want to encode is defined as

$$\begin{aligned} node[x] &= var(x) \\ node[x\ ?] &= app(x, node[?]) \\ node[(?)] &= node[?] \\ node[(?_1)\ ?_2] &= app(node[?_1], node[?_2]) \\ node[\mathbf{fun}\ x \Rightarrow ?] &= abs(x, node[?]) \end{aligned}$$

where $?, ?_1, ?_2 \in \Sigma^*$ (i.e. they match any sequence of input symbols).

The main usage of this stack is to store intermediate AST nodes during the processing of an input string.

Example 3.6. The following are examples where we need to store more than one intermediate AST node. We denote a sequence of input symbols that form a subexpression by e_1, e_2.

1. For prefix $(e_1 \ldots$, the usage of e_1 depends on the following symbol. For example, if the following symbol is x, then we build an *app* node; if it is), then we do not build any new node. As we can not determine how $node[e_1]$ should be used, we store $node[e_1]$ in the stack.
2. For prefix $(\mathbf{fun}\ x \Rightarrow e_1 \ldots$, the usage of x and e_1 depends on the following symbol. For example, if it is x, then we build an *app* node; if it is), we build an *abs* node. Therefore, we store $x, node[e_1]$ in the stack.
3. For prefix $(e_1\ \mathbf{fun}\ x \Rightarrow e_2 \ldots$, the usage of e_1, x, and e_2 depends on the following symbol. For example, if it is x, then we build an *app* node; if it is), we build an *abs* node and then build an *app* node. Therefore, we store $node[e_1], x, node[e_2]$ in the stack.

As this example shows, when a ")" is processed, we may need recursive node building (the amortized time complexity is still linear).

The AST builder amdpa is shown in Table 3. The main difference between Table 2 and Table 3 is that we added one component to the ampda stack vector and refined the rules in column ")". We define ELIA type constructors `x`, `var`, `app`, `abs`, `A`, `Aprimes` to represent x, var, app, abs, A, A'. We show the encoding of the rules

$$\text{Initial Configuration} = 0 \begin{bmatrix} \bot \\ \bot \end{bmatrix}$$

$$\text{Acception Configurations} = \{1 \begin{bmatrix} \bot \\ \bot e_1 \end{bmatrix}\}$$

	x	**fun**	$\Rightarrow$	(
0	$\begin{bmatrix} \\ \end{bmatrix} /1 \begin{bmatrix} \\ var(x) \end{bmatrix}$	$\begin{bmatrix} \\ \end{bmatrix} /2 \begin{bmatrix} \\ \end{bmatrix}$		$\begin{bmatrix} \\ \end{bmatrix} /0 \begin{bmatrix} P \\ A \end{bmatrix}$
1	$\begin{bmatrix} \\ e_1 \end{bmatrix} /1 \begin{bmatrix} \\ app(e_1, x) \end{bmatrix}$	$\begin{bmatrix} \\ \end{bmatrix} /2 \begin{bmatrix} \\ \end{bmatrix}$		$\begin{bmatrix} \\ \end{bmatrix} /0 \begin{bmatrix} P \\ A' \end{bmatrix}$
2	$\begin{bmatrix} \\ \end{bmatrix} /3 \begin{bmatrix} \\ x \end{bmatrix}$			
3			$\begin{bmatrix} \\ \end{bmatrix} /0 \begin{bmatrix} \\ \end{bmatrix}$	

	)		
0			
1	(1)	$\begin{bmatrix} \\ xe_1 \end{bmatrix} /1 \begin{bmatrix} \\ abs(x, e_1) \end{bmatrix}$	$*$
	(2)	$\begin{bmatrix} \\ var(x)e_1 \end{bmatrix} /1 \begin{bmatrix} \\ app(var(x), e_1) \end{bmatrix}$	$*$
	(3)	$\begin{bmatrix} \\ abs(x, e_1)e_2 \end{bmatrix} /1 \begin{bmatrix} \\ app(abs(x, e_1), e_2) \end{bmatrix}$	$*$
	(4)	$\begin{bmatrix} \\ app(e_1, e_2)e_3 \end{bmatrix} /1 \begin{bmatrix} \\ app(app(e_1, e_2), e_3) \end{bmatrix}$	$*$
	(5)	$\begin{bmatrix} P \\ Ae_1 \end{bmatrix} /1 \begin{bmatrix} \\ e_1 \end{bmatrix}$	$\triangleleft$
	(6)	$\begin{bmatrix} P \\ e_1 A' e_2 \end{bmatrix} /1 \begin{bmatrix} \\ app(e_1, e_2) \end{bmatrix}$	$\triangleleft$
2			
3			

Table 3. AST Builder Transition Table: e_1, e_2, e_3 are variable symbols

in column ")" only. Most other rules can be encoded in a similar fashion to the parser encoding. We use different color boxes to mark different components of the encoding: green – Stack 1, blue – Stack 2, red – whether another rule needs to be applied.
Transition Rule (1)

```
implicit <V1,V2,e1>
  G1r<C1< V1 , push<e1,push<x,V2>> >,C>
  r1(G1r<C1< V1 , push<abs<x,e1>,V2> >,C> p)
```

Transition Rule (2)

```
implicit <V1,V2,e1>
  G1r<C1< V1 , push<e1,push<var<x>,V2>> >,C>
  r2(G1r<C1< V1 , push<app<var<x>,e1>,V2> >,C> p)
```

Transition Rule (3)

```
implicit <V1,V2,e1,e2>
  G1r<C1< V1 , push<e2,push<abs<x,e1>,V2>> >,C>
  r3(G1r<C1< V1 , push<app<abs<x,e1>,e2>,V2> >,C> p)
```

Transition Rule (4)

```
implicit <V1,V2,e1,e2,e3>
  G1r<C1< V1 , push<e3,push<app<e1,e2>,V2>> >,C>
  r4(G1r<C1< V1 , push<app<app<e1,e2>,e3>,V2> >,C> p)
```

Transition Rule (5)

```
implicit <V1,V2,e1>
  G1r<C1< push<P,V1> , push<e1,push<Aprime,V2>> >,C>
  r5(G1r<C1< V1 , push<e1,V2> >,0> p)
```

Transition Rule (6)

```
implicit <V1,V2,e1,e2>
  G1r<C1< push<P,V1> ,
     push<e2,push<Aprime,push<e1,V2>>> >,C>
  r1(G1r<C1< V1 , push<app<e1,e2>,V2> >,0> p)
```

3.3.3 Type Checker

We extend our AST builder with a type checker. We introduce two new stacks with signatures $\{x : 0, fun : 2, ty : 2\}$ and $\{x : 0, fun : 2\}$, respectively. Intuitively, the first new stack stores the typing environment, and the second new stack stores the types of the AST nodes in the stack used by the AST builder. To follow convention, we write the function symbol fun as an infix "$\rightarrow$", and the function symbol ty as an infix ":". The general typing partial function $type[-, -]$ that we want to encode is defined as

$$\begin{aligned}
type[var(x), \mathcal{E}] &= lookup[\mathcal{E}, x] \\
type[app(e_1, e_2), \mathcal{E}] &= \tau_2 \\
&\quad \text{if } type[e_1, \mathcal{E}] = \tau_1 \rightarrow \tau_2 \\
&\quad \text{and } type[e_2, \mathcal{E}] = \tau_1 \\
type[abs(x, e), \mathcal{E}] &= \tau_1 \rightarrow \tau_2 \\
&\quad \text{if } type[e, push(x : \tau_1, \mathcal{E})]
\end{aligned}$$

$$\text{Initial Configuration} = 0 \begin{bmatrix} \bot \\ \bot \\ \bot \\ \bot \end{bmatrix}$$

$$\text{Acception Configurations} = \{1 \begin{bmatrix} \bot \\ \bot e_1 \\ \bot \\ \bot \tau_1 \end{bmatrix}\}$$

	x	**fun**	$\Rightarrow$	(
0	$st_3 =_{\tau_1} v : \tau_1 \ldots,$ $\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /1 \begin{bmatrix} \\ var(x) \\ \\ \tau_1 \end{bmatrix}$	$\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /2 \begin{bmatrix} \\ \\ \\ \ \end{bmatrix}$		$\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /0 \begin{bmatrix} P \\ A \\ \\ \ \end{bmatrix}$
1	$st_3 =_{\tau_1} x : \tau_1 \ldots,$ $\begin{bmatrix} \\ e_1 \\ \\ \tau_1 \to \tau_2 \end{bmatrix} /1 \begin{bmatrix} \\ app(e_1, x) \\ \\ \tau_2 \end{bmatrix}$	$\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /2 \begin{bmatrix} \\ \\ \\ \ \end{bmatrix}$		$\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /0 \begin{bmatrix} P \\ A' \\ \\ \ \end{bmatrix}$
2	$\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /3 \begin{bmatrix} \\ x \\ x : \tau_1 \\ \ \end{bmatrix}$			
3			$\begin{bmatrix} \\ \\ \\ \ \end{bmatrix} /0 \begin{bmatrix} \\ \\ \\ \ \end{bmatrix}$	

	)
0	
1	(1) $\begin{bmatrix} \\ xe_1 \\ x : \tau_1 \\ \tau_2 \end{bmatrix} /1 \begin{bmatrix} \\ abs(x, e_1) \\ \\ \tau_1 \to \tau_2 \end{bmatrix}$ $*$ (2) $\begin{bmatrix} \\ var(v)e_1 \\ \\ \tau_1 \to \tau_2\ \tau_1 \end{bmatrix} /1 \begin{bmatrix} \\ app(var(v), e_1) \\ \\ \tau_2 \end{bmatrix}$ $*$ (3) $\begin{bmatrix} \\ abs(v, e_1)e_2 \\ \\ \tau_1 \to \tau_2\ \tau_1 \end{bmatrix} /1 \begin{bmatrix} \\ app(abs(v, e_1), e_2) \\ \\ \tau_2 \end{bmatrix}$ $*$ (4) $\begin{bmatrix} \\ app(e_1, e_2)e_3 \\ \\ \tau_1 \to \tau_2\ \tau_1 \end{bmatrix} /1 \begin{bmatrix} \\ app(app(e_1, e_2), e_3) \\ \\ \tau_2 \end{bmatrix}$ $*$ (5) $\begin{bmatrix} P \\ Ae_1 \\ \\ \ \end{bmatrix} /1 \begin{bmatrix} \\ e_1 \\ \\ \ \end{bmatrix}$ $\triangleleft$ (6) $\begin{bmatrix} P \\ e_1 A' e_2 \\ \\ \tau_1 \to \tau_2\ \tau_1 \end{bmatrix} /1 \begin{bmatrix} \\ app(e_1, e_2) \\ \\ \tau_2 \end{bmatrix}$ $\triangleleft$
2	
3	

Table 4. Type Checker Transition Table: $v, e_1, e_2, e_3, \tau_1, \tau_2$ are variable symbols

where the partial function *lookup* is defined as

$$\begin{array}{rcl} lookup[push(x:\tau,\mathcal{E}),x] & = & \tau \\ lookup[push(y:\tau,\mathcal{E}),x] & = & lookup[\mathcal{E},x] \\ & & \text{if } y \neq x \end{array}$$

The type checking amdpa is shown in Table 4. The main difference between Table 3 and Table 4 is that we added two components to the ampda stack vector and added lookup rules in column x.

We define ELIA type constructors `fun`, `ty` to represent fun, ty. We only show how to encode rules in cell $(0, x)$. Most other rules can be encoded in a similar fashion as we did before. We use different color boxes to mark different components of the encoding: green – Stack 1, blue – Stack 2, purple – Stack 3, orange – Stack 4, yellow – the type pattern to look up.
Generic Lookup Rule 1

```
implicit <T,V> L< push<T,V> , T > r0()
```

Generic Lookup Rule 2

```
implicit <V,T1,T2>
   L< V , T1 > r2(T2!=T1 q,L< push<T2,V> , T1 > p)
```

Transition Rule 3 in cell $(0, x)$

```
implicit <V1,V2,V3,V4,t1>
  G0v<C0< V1 , V2 , V3 , V4 >,C, t1 >
  r3(G0v<C0< V1 , push<var<x>,V2> , V3 , push<t1,V4> >,
    0, t1 > p)
```

The method `x` looks like the following.

```
interface C0<V1,V2,V3,V4> {
  <T,t1> T x(|
    L<V3,ty<x, t1 >> p0,
    G0v<T,S, t1 >->G0v<C0<V1,V2,V3,V4>,C, t1 > p);
}
```

3.4 Integration with Interpreters

With the AST builder and type checker, we are capable of building a type that represents a well-typed (in the object language type system) AST. Using ACP constructors, we can convert this type to ACP values that can be executed at runtime.

1. We define necessary auxiliary classes for interpreting object language ASTs. We defined the following interface:

   ```
   interface Val<S> {
     <T> Val<app<S,T>> app(T o);
   }
   interface Env {
     <V,e> Env push(V x, Val<e> o);
     <V> Val<Var<V>> lookup(V x);
   }
   ```

2. For each AST node constructor f, we define an ACP constructor that converts types representing AST nodes constructed from f to ACP values the represents the computation of the AST nodes. The basic idea is that given an ELIA type e_1 representing an AST, we ask the compiler to construct an ACP value of type `Env->Val<`e_1`>`. The following rules recursively deconstruct e_1 and perform ACP value construction on its subterms. For the *app* node, we define the following ACP constructor:

   ```
   implicit <e1,e2> Val<app<e1,e2>> Rapp(
     Env->Val<e1> a, Env->Val<e2> b, Env env) {
     return a(env).app(b(env));
   }
   ```

   For the *abs* node, we define the following ACP constructor:

   ```
   implicit <V,e1> Val<abs<V,e1>> Rabs(
     Env->Val<e1> a, V vari, Env env) {
     return new Val<abs<V,e1>>() {
       <e2>
       Val<app<abs<V,e1>,e2>> app(Val<e2> o) {
         return a(env.push(vari,o));
       }
     };
   }
   ```

   For the *var* node, we define the following ACP constructor:

   ```
   implicit <V> Val<var<V>> Rvar(
     V vari, Env env) {
     return env.lookup(vari);
   }
   ```

   For the x node, we replace the definition of type constructor `x` with the following class to make it a singleton type:

   ```
   class x {
     private static x _x = new x();
     private x() {}
     public x getx() { return _x; }
   }
   ```

   and define the following singleton class and ACP constructor:

   ```
   implicit x Rv() {
     return x.getx();
   }
   ```

3. Finally, we define a interpretor function that ties the ACP constructors defined above to our AST builder.

   ```
   class C1<V1,V2,V3,V4> {
     <e,t> Val<e> run(|
     V1==bot p0,V2==push<e,bot> p1,
     V3==bot p2,V4==push<t,bot> p3,
     Env->e r) {
       return r(new Env());
     }
   }
   ```

We show an example of embedding STLC programs using our encoding.

Example 3.7. In this example, we assume that our ampda has two more input symbols y and z that behave like the input symbol x. Assume that we have an object `prog` of type `C0<bot,bot,bot,bot>`. The STLC program **fun** $x \Rightarrow x$ can be embedded as the following COMIC:

```
prog.fun().x().arrow().x().run()
```

The constructed ACP value for the ACP `r` in method `run` is

```
λenv1.Rabs(Rx,λenv2.Rvar(Rx,env2),env1)
```

where we use the λ notation to represent functions. An expression $\lambda x.e$ denotes a unary function that returns the value of e given a value of x. The STLC program **fun** $x \Rightarrow$ **fun** $y \Rightarrow y\ x$ can be embedded as the following COMIC:

```
prog.fun().x().fun().y().arrow().y().x().run()
```

The constructed ACP value for the ACP `r` in method `run` is

```
λenv1.Rabs(Rx,λenv2.Rabs(Ry,λenv3.Rapp(
  λenv4.Rvar(Ry,env4),
  λenv5.Rvar(Rx,env5),env3),env2),env1)
```

The STLC program (**fun** $y \Rightarrow y$) (**fun** $x \Rightarrow x$) can be embedded as the following COMIC:

```
prog.l().fun().y().arrow().y().r()
    .l().fun().x().arrow().x().r().run()
```

The constructed ACP value for the ACP `r` in method `run` is

```
λenv1.Rapp(
  λenv2.Rabs(Ry,λenv3.Rvar(Ry,env3),env2),
  λenv4.Rabs(Rx,λenv5.Rvar(Rx,env5),env4),env1)
```

Our interpretor is very similar to the "final approach" [4], which allows an embedded program to be written once and evaluated in many different ways. Our approach allows the embedded program to be written once, represented internally in many different ways (such as final or initial), and evaluated in many different ways.

4. Discussion

Improving Readability ELIA allows using operators as method names and omitting dots and parentheses if the method has no parameters, which can be used to improve the readability of embedded programs. For example, if we replace the method name `arrow` in the previous example with method name `=>`, then we can embed $\mathbf{fun}\, y \Rightarrow \mathbf{fun}\, x \Rightarrow y\, x$ as `prog fun y => fun x => y x run`.

Error Messages When using a general purpose programming language as the host language for an embedded DSL, error messages produced by the host language compiler are usually less relevant to the embedded DSL. As shown in Section 2.4, ELIA has a feature that allows programmatically defining customized error messages to be output when the compiler fails to construct ACP values. Using this feature, we can define error messages in our encoding that are more relevant to the embedded DSL.

Extending/Restricting Our Framework In the framework introduced in this paper, we assume that every input symbol is mapped to a method for clarity. An impact of this simplification is that we can only encode a finite number of constants or variables in our framework. To avoid this limitation, we can extend our ampda so that the input is a string of terms generated from a signature. Using this extension, the methods in the encoding will have regular Java parameters and will allow an infinite number of constants and variables.

If we restrict ampdas to have one stack and combine stack symbols with states, then our framework reverts back to our previous framework [34, 35], which can be implemented in Java.

Encoding Generation Manually writing the ELIA code that encodes a parser, type checker, AST builder, and interpreter of a complex DSL can be very tedious. Our framework enables automatic generation of this code from a DSL specification. In our previous work, we created a tool that can be used to automatically generate code from DSL specifications written in a high-level specification language [35]. Although our code generator only implements a restricted version of our general framework tailored to the static programming capability of Java, it can be easily modified to generate code for ELIA and implement our full framework.

Challenges in Designing ELIA There are some challenges in designing ELIA. The first challenge is to avoid non-termination of compilation. There are a few techniques that can be applied to detect static programs that diverge. Currently, we use a simple order-based termination proving technique. For complex ACP constructors and ACPs, we can use the "transformational" approach in automated termination analysis for logic programs [27] (and references therein), which translates a logic program into a term rewriting system (TRS) and applies termination analysis on the TRS. Another challenge is formalizing the extension. It turns out that ELIA can be easily formalized by extending the FGJ calculus [11] with ACPs, as the interaction between ACPs and the type system of Java is minimal. A third challenge is to make ELIA useful beyond the scope of language embedding. The generality of ELIA allows us to use it in other scenarios such at automatic type conversion and enforcing user-defined static properties of methods.

5. Related Work

DSLs Most of the research on embedded DSLs is in the functional setting. There are proposals for embedding typed languages into typed host languages, based on GADT [13, 32], dependent types [7, 25, 33], ordinary functions [4], or higher order abstract syntax [26]. Research is usually focused on creating tagless interpreters, which reduce the overhead in embedded DSL implementations. Implementing embedded DSLs in OO/functional programming languages [10] and pure OO programming langauges [34, 35] is a less extensively studied area of research. Our examples show that COMIC embedding can be used together with tagless interpreters to implement embedded DSLs in an OO setting.

Many programming languages support defining embedded DSLs to some extent. However, when designing embedded DSLs in these languages, the designers need to deal with restrictions of the host language. Many DSL systems provide meta languages for the specific domain of DSL definition [20, 30]. In these tools, the meta languages are themselves domain specific languages. Pattern languages, such as OMeta [31] and π [17], allow defining patterns and their semantics, and can also be used as meta languages for embedded DSLs.

OO Programming ELIA is based on Java [8]. Some features of ELIA, such as flexible syntax, were inspired by Scala [23]. However, the most important features of ELIA, such as ACPs and customizable error messages, are not in Scala. The equality type for ACPs was inspired by a similar typing constraint proposed for $C\sharp$ [15]. ACP types differ from typing constraints in that they are not only constraints that need to be solved, but also specifications from which ACP values are inferred. Predicate dispatch [5, 6, 21, 24, 29] is a mechanism for dynamically determining the code to be executed upon a method invocation based on user-defined predicates. ACP types do not interfere with dynamic method dispatching and do not have runtime overhead.

Logic Programming Concepts such as "cut" and backward chaining are used in Prolog [28] and automatic theorem proving. However, in their applications, a proof that a query is satisfiable is usually an addition to the result of the query; while in ELIA, the proof is essential for constructing ACP values.

Type classes [9] 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 embedded DSLs. ELIA incorporates logic programming into an OO programming language.

Some functional programming languages support explicitly the notion of proofs [7] (and references therein) through dependent types. A dependently typed language usually either has a type language that is separate from 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 dependent type theory [19]. 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 a practical host language for embedded DSLs, if it has a strong type inference algorithm.

6. Conclusion

Our paper introduced a powerful framework for embedding DSLs into OO programming languages. Unlike other approaches, which embed object language programs as nested calls to functions or data constructors, our framework embeds object language programs as a chain of method invocations. We used ampdas, our variant of pdas, as an intermediate devices for representing parser, AST builders, and type checkers in a uniform fashion. We demonstrated our framework by using an example of encoding the STLC in ELIA, our extension to Java with ACPs. This example shows that the COMIC embedding can be used to embed DSLs of other programming paradigms into OO host languages. The ideas of ACPs and COMIC embedding could also be extended to host languages of other programming paradigms, given proper encoding. We hope that in the future, we can extend our framework to other programming languages and paradigms.

References

[1] http://code.google.com/apis/protocolbuffers/.

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

[3] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. *Context-Free Language and Pushdown Automata*, volume 1 Word, Language, Grammar of *Handbook of Formal Languages*, chapter 3, pages 111–174. Springer, 1997.

[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] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] Andrzej Indrzejczak. *Natural Deduction, Hybrid Systems and Modal Logics*. Springer, 2010.

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

[14] Donald Kalish, Richard Montague, and Gary Mar. *Logic: Techniques of Formal Reasoning Second Edition*. Oxford University Press, 1980.

[15] 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.

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

[17] 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.

[18] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

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

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

[21] 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.

[22] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

[24] 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.

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

[26] 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.

[27] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Trans. Comput. Logic*, 11(1):1–52, 2009.

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

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

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

[31] 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.

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

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

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

[35] Hao Xu. Erilex: An embedded domain specific language generator. In *TOOLS 2010*, 2010.