Foreign Language Interfaces by Code Migration

Shigeru Chiba
The University of Tokyo
Graduate School of Information Science and Technology
Tokyo, Japan
chiba@acm.org

Abstract

A foreign function interface (FFI) is a classical abstraction used for interfacing a programming language with another *foreign* language to reuse its libraries. This interface is important for a new (or non prevailing) language because it lacks libraries and thus needs to borrow libraries written in a foreign language when the programmer develops a practical application in that new language. However, a modern library often exploits unique language mechanisms of the implementation language. This makes the use of the library difficult through a simple function call from that new language. This paper presents our approach to this problem. We use an embedded domain specific language (DSL), which is designed to resemble the foreign language, and migrate the DSL code to access to the library written in the foreign language. This paper also presents our framework *Yadriggy* for developing the DSL from Ruby to a foreign language environment. The framework supports DSL-specific syntax checking for the migrated DSL code.

CCS Concepts • **Software and its engineering → Runtime environments**; *Domain specific languages.*

Keywords foreign function interface, polyglot programming, library, Ruby, Python

ACM Reference Format:
Shigeru Chiba. 2019. Foreign Language Interfaces by Code Migration. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357765.3359521

1 Introduction

A programming language without a rich set of libraries will not be used for practical application development. Providing a wide range of libraries is a significant issue for programming language developers who want to make their languages practical. Hence some programming languages such as Java, Python, and Ruby provide a mechanism for calling library functions written in the C language. Since there are a wide variety of C libraries and operating system services are often provided through C libraries, bridging to C functions is a cost-effective approach to widening the coverage of libraries. Furthermore, the C language is relatively simple compared to modern languages and a function call in C is an abstraction available in most programming languages. A function-based bridging mechanism to C, which is often called a foreign function interface (FFI), can be seamlessly embedded in those languages. Although passing a pointer value as an argument needs a somewhat complex trick to implement, Proxy pattern [12] using reflection [35] is a well known solution.

A number of useful libraries are, however, being implemented in other languages than the C language. For example, Python is a popular language today when implementing a library for machine learning and scientific computing. TensorFlow [16], PyTorch [1], and Matplotlib [20] are examples of popular Python libraries. Since the application programming interfaces (API) of these libraries exploit language features unique to Python, the client code of those libraries is aware of these features. This fact is a hindrance when using the libraries from a different language since its FFI has to support Python-like function/method calls but these calls might not be naturally expressed in that different language.

This paper presents that code migration is an appropriate abstraction for the interfaces to foreign-language libraries. When accessing a library written in a foreign language in our approach, a host language program sends a code block to the foreign language, which executes the code block accessing the library and returns the result back to the host language program. The host language program does not access the library through a function call. The migrated code block is written in a domain-specific language (DSL) embedded in the host language. It borrows the syntax from the host language but it is semantically more similar to the foreign language. This DSL allows accesses to the unique features of the foreign language while it also manages to pass data between the host and foreign languages. The DSL helps the programmers avoid several pitfalls that they may encounter when they access a foreign-language library according to the library's tutorial written for the foreign-language programmers.

GPCE '19, October 21–22, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359521

To explore this approach, we have developed *Yadriggy*[1], a framework in the Ruby language. It is used to implement an interfacing system from Ruby to the libraries written in a foreign language, such as Python. Our challenge is to realize the approach without modifying Ruby or its programming environment. With this aim, the framework assumes that the syntax of the DSL is a subset of Ruby and only its semantics is uniquely designed. The DSL will be regarded as a variant of Ruby extended with foreign-language features, in other words, as a language borrowing the syntax from Ruby but the semantics from the foreign language. The framework provides a facility for dynamically extracting an abstract syntax tree (AST) for a code block given in the form of lambda expression or method object. The framework also provides a syntax checker to examine the extracted AST consists of only the selected syntactic forms for the DSL. It is a key component to deal with a code block in a language with complex syntax like Ruby. The developer of the interfacing system would have to write tedious error-check code if the syntax checker is not available. Restricting the available syntax in the DSL would also reduce the amount of work by the developer. Furthermore, the syntax checker is used to tag a particular shape of subtree of the AST. This tag can be used later, for example, during code generation.

In the rest of this paper, Section 2 presents our motivating examples. Section 3 proposes our framework and Section 4 shows how the framework is used. Section 5 presents related work and Section 6 concludes this paper.

2 Are Foreign Function Interfaces Adequate?

Ruby is a programming language that is popular for developing web services [15]. Python is popular for scientific computing and machine learning and hence there are several well-known mature libraries in those application domains, such as TensorFlow [16] and Matplotlib [20]. Thus, it seems a good idea to enable Ruby programs to access these Python libraries through a foreign function interface (FFI). In Ruby, PyCall [22] is available as an FFI library bridging between Ruby and Python. PyCall is a Ruby port of the library with the same name for the Julia language [37].

With PyCall, we can write the following Ruby code:

```
plt = PyCall.import_module('matplotlib.pyplot')
plt.plot([1, 3, 2, 5])
plt.show()
```

This draws a line graph by using the Matplotlib library in Python. The variable `plt` refers to a proxy object. The call to the `plot` method on `plt` results in the *remote* method invocation on the `matplotlib.pyplot` module in Python. Here, *remote* means the outside of the Ruby virtual machine.

[1]Available from https://github.com/csg-tokyo/yadriggy or Zenodo [5].

The call to `show` on `plt` also results in the *remote* invocation of `show`. The code above is equivalent to this Python code:

```
import matplotlib.pyplot as plt
plt.plot([1, 3, 2, 5])
plt.show()
```

The two code snippets look very similar. Only the difference is that `plt` in Python is a shorthand of the module name `matplotlib.pyplot`.

PyCall is an FFI library with elaborate design. It exploits the syntactical similarity between Ruby and Python and enables Ruby programmers to naturally call a Python function. They can call a Python function mostly by just copying a code snippet written in Python from the tutorial.

However, a few blog posts point out its pitfalls [8, 30]. Suppose that we want to write the following Python code:

```
from collections import deque
dq = deque([2, 3, 5])
```

The second line might look like a function call but it is a constructor call; it creates a new instance of the `deque` class. In Ruby with PyCall, hence, we must write the following code:

```
pyfrom 'collections', import: :deque
dq = deque.([2, 3, 5])
```

Note that the dot operator follows `deque`. Alternatively, we can write:

```
pyfrom 'collections', import: :deque
dq = deque.new([2, 3, 5])
```

Now the second line calls the `new` method on the `deque` class so that an instance will be created. If we are aware that `deque` in Python is a constructor call, we would be able to correctly write the corresponding Ruby code above. However, if we are careless, we might forget to write the dot or `new` operator after `deque` and encounter a runtime error.

Another example is a keyword argument in Python. Suppose that we want to pass the keyword argument `lw` (line width) to the `plot` function. In Python, we would write:

```
import matplotlib.pyplot as plt
plt.plot([1, 3, 2, 5], lw=5)
plt.show()
```

If we execute the second line in Ruby with PyCall, the argument `lw=5` is interpreted by the Ruby interpreter as the assignment to the variable `lw`. The arguments are evaluated in the context of Ruby and only the resulting values are sent to Python by PyCall. Hence, the Python interpreter attempts to execute the call `plt.plot([1, 3, 2, 5], 5)` and then we will see a runtime error. To avoid this misinterpretation, we must write the following code in Ruby:

```
plt = PyCall.import_module('matplotlib.pyplot')
plt.plot([1, 3, 2, 5], lw: 5)
plt.show()
```

In Ruby with PyCall, we must use the Ruby syntax for keyword arguments, `lw:5`. Again, if we are aware of the difference between Python and Ruby with respect to keyword arguments, we would not make such a mistake. However, we cannot use a Python library from Ruby by just copying a code snippet from the tutorial of that library although the code snippet looks syntactically correct in Ruby as well as Python.

The final example is a dictionary. The following Python program is a simple example for running TensorFlow.

```
import tensorflow as tf
sess = tf.Session()
x_data = tf.placeholder(tf.float32)
expr = tf.multiply(x_data, x_data)
r = sess.run(expr, feed_dict={x_data: 2.0})
```

The second line is a constructor call and the last line is a call with a keyword argument. Hence the following Ruby program might seem to correctly work:

```
tf = PyCall.import_module('tensorflow')
sess = tf.Session.new()
x_data = tf.placeholder(tf.float32)
expr = tf.multiply(x_data, x_data)
r = sess.run(expr, feed_dict: {x_data: 2.0})
```

Unfortunately, this raises a runtime error because the interpretation of the dictionary literal `{x_data: 2.0}` is different between Python and Ruby. In Python, `x_data` is regarded as an expression and thus the created dictionary maps the value of the variable `x_data` to `2.0`. In Ruby, `x_data` is regarded as a symbol and thus the created dictionary maps `"x_data"` to `2.0`. To fix this problem, the last line must be as follows:

```
r = sess.run(expr, feed_dict: {x_data => 2.0})
```

The literal `{x_data => 2.0}` in Ruby is equivalent to `{x_data: 2.0}` in Python.

The direct causes of these pitfalls are minor differences between Python and Ruby. The first example is due to the syntactic difference in constructor calls. The second one is due to the syntax of keyword arguments. The third one is due to the syntax of dictionaries. The programmers might not make a mistake due to these rather minor differences if they are sufficiently careful when writing foreign function calls between Ruby and Python. A function call is, however, an abstraction that tends to provoke such a mistake. The called function is executed in Python through PyCall but its function name and its arguments are computed in Ruby.

As we mentioned above, the following Ruby code causes a runtime error because no dot or `.new` follows `deque`:

```
pyfrom 'collections', import: :deque
dq = deque([2, 3, 5])
```

Although we might want to consider the second line as the Python code embedded in Ruby code, the function name `deque` is first evaluated as Ruby code. It has to result in a Ruby object representing the Python `deque` class. The object may receive `.()` and `.new()` but not `()` as a message. For the other examples, we observe similar error-prone code boundaries between Ruby and Python. We might want to consider the following lines:

```
plt.plot([1, 3, 2, 5], lw=5)
```

```
r = sess.run(expr, feed_dict: {x_data: 2.0})
```

as embedded Python code, but the arguments are evaluated in Ruby. Only the bodies of `plot` and `run` are executed in Python.

PyCall also allows a Ruby program to access a Python library by string embedding as well as a foreign function call. In this approach, the whole Python source code is encoded as a string object in Ruby and it is passed to PyCall for execution. For example,

```
lst = [2, 3, 5]
dq = PyCall::eval("deque(#{lst})")
```

it first constructs Python source code as a string object by string interpolation (or formatting). `lst` is evaluated by Ruby and the resulting value is embedded. Then the string object is given to `eval` to be executed by the Python interpreter. In this approach, the code boundary between Ruby and Python is more explicit than in a foreign function call. The code surrounded with double quotes is Python code except the Ruby code within `#{` and `}`. We do not need to add an extra dot or `.new` after `deque`.

However, the string-embedding approach is still error-prone [9]. Syntax highlighting will not be applied to the source code encoded as a string literal. The string interpolation composes the source code on a lexical level and thus it often leads to syntax errors or security holes such as SQL injection and cross-site scripting attacks. The programmers must be still aware of the code boundaries between the languages.

3 Code Migration Substitutes for Foreign Function Interfaces

To give an alternative to the interface based on function calls, we discuss an approach based on migrating code block to a foreign environment, for example, from the Ruby interpreter to the Python interpreter (or between the virtual machines). Although string embedding is one of the techniques for this approach, we present another technique to mitigate drawbacks of string embedding. We also present *Yadriggy*, our new framework for Ruby, which provides a basic facility for implementing our technique in Ruby.

3.1 Overview

Yadriggy provides a basic facility for implementing an interfacing system to access a library written in another language from Ruby. We below call this interfacing system *a foreign language interface* because its interface is not a function-call basis. This term is also seen in the Prolog family [39].

The foreign language interface built on Yadriggy takes a code block in the form of lambda expression or method object. Then it migrates the whole code block to the foreign language environment for execution as the string-embedding approach does. For example, the interfacing system would be used as the following:

```
lst = [2, 3, 5]
dq = run_python { deque(lst) }
```

In Ruby, the code block surrounded with { and } is regarded as a lambda expression (or a Proc object).[2] It is passed to run_python, which is a method provided by the interfacing system. The code boundary between Ruby and Python is much simpler than a function-call basis interfacing system. Note that a dot or .new does not follow deque. Since run_python migrates the whole expression deque(lst) to the Python interpreter, the expression is interpreted as a constructor call.

To migrate the code block, run_python first obtains an abstract syntax tree (AST) for the source code of that code block. Obtaining an AST is supported by Yadriggy. Then run_python generates the Python code sent to Python. It identifies free variables in the code block and sends their values to Python as well as the code. For example, the variable lst is a free variable and it refers to a Ruby array denoted as [2, 3, 5]. Sending a copy of this array to Python is the responsibility of run_python, the interfacing system, so that the migrated code can access that array. The programmer does not have to manually embed the array into the migrated code by string interpolation. This would make the interfacing system less error-prone than the string-embedding approach.

Our idea is to use a normal lambda expression for expressing a code block passed to the foreign language environment. We do not allow syntax extensions for describing the code block. The programmers cannot enjoy domain-specific syntax but they can work with a normal Ruby programming environment since we do not modify the normal Ruby interpreter, which does not enable user-defined syntax extension [9, 18, 26] or reader macros [36].

We believe the use of only Ruby's syntax is not a serious problem. Ruby's syntactic flexibility allows us to write Ruby code that looks like foreign language code or closely resembles it. For example, as we have seen, the expression {x_data: 2.0} in Python is valid Ruby code except the semantics. Slice notation in Python such as a[i:j] is not valid syntax in Ruby. In this case, we can pick similar valid syntax in Ruby such as a[i..j] and implement the interfacing system so that it will map a[i..j] to a[i:j]. The programmer must be concerned about this syntactic difference, but we believe that she would not be badly confused since slice notation is not available in Ruby.[3] A problematic case is only the syntax valid in both languages with different semantics.

Note that the migrated code block is written in normal Ruby but it does not have to be interpreted with Ruby's original semantics. It can be interpreted as Python code or the code written in an original DSL, which is embedded in Ruby but borrows only its syntax. Furthermore, the language for the migrated code does not have to support the full set of Ruby's syntax. The language's syntax is designed by restricting Ruby's rich syntax, not by defining the new syntax from scratch.

[2]Similar syntax is also seen in Scala.

3.2 Extracting a Syntax Tree

To extract an AST for the given code block as a lambda expression, Yadriggy provides the reify method. It takes a lambda expression as an argument, finds the source-code location where the lambda expression was constructed, and returns an AST for the source code of that lambda expression. The reify method can also take a Method object and return an AST for the method declaration. A Method object is a metaobject representing a method. It is part of the standard Ruby.

The reify method is similar to classic Lisp macro systems [7, 36, 38] or compile-time reflection [4] but reify does not perform preprocessing or macro expansion at its call site. It rather constructs an AST of the source code located somewhere far from the call site to reify. Therefore, the entry point of the foreign language interface built on our framework, such as run_python shown above, is not a macro function, either; it is a normal method.

If run_python were a macro function, the code block given to it would be transformed into an AST where run_python was called, for example, at this call site:

```
dq = run_python { deque(lst) }
```

Then run_python would return source code to lexically replace the original macro call. Finally, the returned source code would be executed. The definition of the macro function would be something like this:

```
def_macro run_python(ast)
  code = generate_python_code_from(ast)
  return "PyCall::eval('#{code}')"
end
```

Note that this is pseudo code since Ruby does not support macro functions.

On the other hand, run_python using our framework would be defined as follows:

```
def run_python(&block)
  ast = reify(block)
  code = generate_python_code_from(ast)
```

[3]In Ruby, i..j makes a Range object representing an interval from i to j. So a[i..j] in Ruby is also semantically similar to a[i:j] in Python although we have to write a[i...j] (not two but three dots) in Ruby to get the same result as a[i:j].

```
  return PyCall::eval(code)
end
```

The parameter `block` is bound to the lambda expression[4] representing the block argument `{ deque(lst) }`. The AST for this block argument is constructed when `reify` is called, not when `run_python` is called. Then `run_python` generates Python code, migrates, and executes it by the Python interpreter. Finally, `run_python` returns the resulting value.

Constructing an AST for a lambda expression is an advantage of the `reify` method against macro functions. This design is useful because the programmer can define her own method to extend `run_python`. For example,

```
def run_python_with_logging(&block)
  puts "begin Python"
  result = run_python(&block)
  puts "end"
  return result
end
```

This method prints a log message and passes the given block argument to the `run_python` method. If `run_python` is a macro function, `run_python_with_logging` would be also a macro function and its body would include a redundant copy of the body of `run_python` or it would have to perform error-prone string concatenation before nested macro-expansion by `run_python`.

The AST constructed by the `reify` method provides supports for the foreign language interface when identifying a free variable and its value. The AST consists of tree-node objects connected by bidirectional links. Each node object has the `value` method, which returns a runtime value represented by that node. The `value` method on most node objects returns `Undef` (undefined). However, if the tree node represents a literal or a free variable name, it returns the value of that literal or variable. Since the `reify` method constructs an AST at runtime, it also captures the binding environment for the lambda expression or the method. The `value` method accesses this environment to obtain the runtime value. For example,

```
lst = [2, 3, 5]
dq = run_python { deque(lst) }
```

When the `reify` method constructs an AST for the block argument `{ deque(lst) }`, it captures the binding environment for that block argument (*i.e.* a lambda expression). Figure 1 illustrates this AST. The AST recognizes that the free variable `lst` refers to an array denoted as `[2, 3, 5]` and the `value` method returns this array when it is called on the `Identifier` node representing `lst`. On the other hand, the `value` method returns `Undef` when it is called on the `Identifier` node representing `deque`. This is because `deque` is an undefined name in Ruby. If `deque` referred to a Ruby method visible from the block argument, the `value` method would return that method.

[4]More precisely, `block` is bound to a `Proc` object. Although `Proc` is not equivalent to a lambda expression, we treat them as interchangeable for brevity.

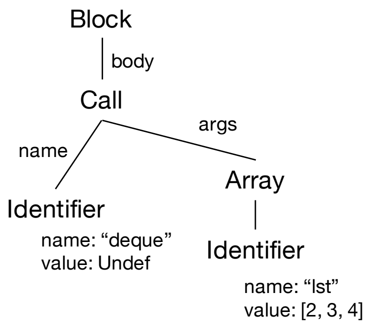

Figure 1. The abstract syntax tree

The current implementation of the `reify` method calls the `source_location` method on the `Proc` or `Method` object given to `reify`. `source_location` is part of the standard Ruby. Since `source_location` returns the source file name, the line number, and the column position, the `reify` method first parses the source file and extracts the AST for that `Proc` or `Method` object. Because the `ripper` parser used by the `reify` method provides incomplete information on the token locations, the `reify` method may fail to extract a correct AST when multiple lambda expressions appear in the same line. To address this limitation, we have to reimplement the `reify` method by using the `RubyVM::AbstractSyntaxTree` module newly introduced by Ruby 2.6.

3.3 Syntax Checking

The foreign language interface built on Yadriggy should check the migrated code is syntactically valid or not before it executes the migration. Since the migrated code is a lambda expression passed to the `reify` method and it is written in the normal Ruby syntax, if the migrated code includes a syntax error, the Ruby interpreter will detect it when constructing the lambda expression before running the foreign language interface.

However, the migrated code will be written in not full-featured Ruby but its subset, or a DSL consisting of only selected forms of expression from Ruby's. For example, the foreign language interface from Ruby to Python would support only the forms compatible with Python's. The migration of some forms of expression might have not been implemented and thus these forms might need to be unavailable in that DSL.

To check whether the migrated code includes only the supported forms of expression, Yadriggy provides a *syntax* checker for the ASTs as well as the `reify` method. Since the `reify` method constructs an AST for the migrated code even when the code includes unsupported forms of expression (as far as it is valid Ruby code), the syntax checker examines that the shape of the AST consists of only the supported forms of expression. It helps implement the foreign language interface because the implementation can assume that the obtained AST always has a valid shape after the checking.

The valid shapes of ASTs are specified in our DSL, which is also built on Yadriggy. For example,

```
syn = define_syntax do
  Binary <= {op: :+ | :-, left: expr,right: expr}
  expr <= Binary | Number
  Block <= { body: expr }
end
ast = ...
puts syn.check(ast) # true if ast is a valid AST
```

`define_syntax` is a method provided by Yadriggy. The do ... end block[5] is passed to `define_syntax` as a lambda expression. The capitalized names `Binary` and `Block` are the class names for AST nodes. The second line reads that a `Binary` node in the AST has `:+` or `:-` (`+` or `-` symbol[6]) for `op` (operator) and an `expr` node for `left` operand and `right` operand, respectively. Here, `op`, `left`, and `right` are accessor methods in the `Binary` class. `expr` is a user-defined node type. It is defined in the third line; it is either `Binary` or `Number`. The forth line reads that a `Block` node has an `expr` node for `body`. Note that `|` is an ordered choice. The `define_syntax` method creates a syntax checker and the `check` method on this checker examines whether the given AST satisfies the rules given to `define_syntax`.

We can regard this checking as syntax checking because the rules indirectly specify the syntax of the code represented by the AST. The rules above correspond to the following syntax rules in PEG-like notation [10]:

```
binary  : expr '+' expr | expr '-' expr
expr    : binary | number
block   : expr
```

Note that `|` is an ordered choice. A difference is that each operand of rule is named as `left`, `right`, and so forth, in our framework. The order of the operands is not specified in our framework since the program has been already parsed according to the normal Ruby syntax.

Each rule given to `define_syntax` takes the following form:

type_name <= *constraint*

The left operand of <= is a type name of AST node. It is either a Ruby class or an arbitrary identifier[7], which is recognized as a user-defined node type. The rule specifies that the AST nodes of this type satisfy the constraint on the right hand side. The syntax checker first visits the root node of the given AST. It attempts to find the rule for the node type. When it does not find the rule, it next attempts to find the rule for the super type. If no rule is found, the checker reports the AST is valid. Otherwise, it checks whether the node satisfies the constraint in the found rule.

The constraints available in the rules are listed in Table 1. If the constraint is a type name, the AST node has to be of that type and also satisfy the rule for that type. The syntax checker recursively finds the rule and checks the constraint. If the constraint is a hash literal such as `{ body: expr }`, the checker checks that the attributes of the AST node satisfy the attribute constraints in Table 2.

When the following hash literal is given:

{ key_1: $value_1$, key_2: $value_2$, ..., key_n: $value_n$ }

for each pair of key_i and $value_i$, the checker checks that the value of the attribute named key_i matches the attribute constraint $value_i$. The value of the attribute key_i is obtained by calling the method named key_i on the AST node. An attribute of the AST node is not tested when the hash literal does not contain the attribute's name as a key. When the attribute constraint is a type name, the checker examines that the attribute's value is an object of that type and it satisfies the constraint for that type, if any.

The attribute constraint may be an array literal. It specifies that the attribute's value is an array satisfying the constraint. The constraints for arrays are listed in Table 3. In this table, `c`, `c0`, `c1`, and `c2` are attribute constraints.

When the given AST node matches a user-defined type, it is tagged as that user-defined type. The tag can be used to identify a particular shape of tree during the tree walking mentioned later. For example, the following rules identify a method-call expression to the `this` method:

```
expr <= this_variable | VariableCall
this_variable <= VariableCall + { name: "this" }
```

Since `|` is an ordered choice, the syntax checker first attempts `this_variable`. The second rule specifies that `this_variable` is an node object of the `VariableCall` class and its `name` attribute is a string `"this"`. Since the most specific user-defined type is attached to the AST node, the AST representing a method call to `this` is tagged as `this_variable`. Otherwise, an AST node of the type `VariableCall` is tagged as `expr`. This would be useful to implement the foreign language interface to C++. Note that `this` is not a reserved keyword in Ruby but it is regarded as a method-call expression on `self` without arguments, `self.this()`.

The rules for the syntax checker of Yadriggy is also used to show the specification of the ASTs obtained by the `reify` method of Yadriggy. The source code of Yadriggy is distributed with the rule sets that match any ASTs obtained by the `reify` method. They are the reference manual that shows the class names for the AST nodes and their attribute names. See Appendix A for the details of the rule sets.

[5]do ... end is equivalent to {...} for making a block argument (*i.e.* a lambda expression passed as an argument).

[6]In Ruby, a name starting with a colon is a symbol. `:id` is a symbol named `id`.

[7]A name starting with a lowercase letter. In Ruby, a class name starts with an uppercase letter.

Table 1. Constraints

constraint	*specification*
`nil`	The AST node is not valid.
type name	The AST nodes is an object of that type and it satisfies the constraint for that type.
hash literal	The attributes of the AST node satisfies the given constraints.
`c1 + c2`	Both constraints `c1` and `c2` are satisfied. (+ has higher precedence than `|`.)
`c1 | c2`	Ordered choice. Either `c1` or `c2` is satisfied.

Table 2. Constraints for node attributes

hash value	*specification*
`nil`	The attribute is `nil` or an empty array.
type name	The attribute is a value of that type.
string literal	The value of the attribute matches that string.
symbol literal	The value of the attribute matches that symbol.
array literal	The attribute is an array and the elements match the constraint.
`( c )`	The attribute constraint `c` is optional. The attribute is `nil` or satisfies `c`.
`c1 | c2`	Ordered choice. The attribute satisfies either the attribute constraint `c1` or `c2`.

Table 3. Constraints for arrays

array literal	*specification*
`c`	The array contains a single element. It matches `c`.
`[]`	An empty array.
`[ c ]`	All the array elements match the attribute constraint.
`[ c0, c1 ]`	The first element matches `c0` and the rest of the elements match `c1`.
`[ c0, c1, c2 ]`	The first two elements match `c0` and `c1`, respectively. The rest of the elements match `c2`.
`[ c0 * c1 ]`	Each array element is an array with two AST nodes.
	The first element of each array element matches `c1` and the second element matches `c2`.

3.4 Tree Walker

Yadriggy also provides a DSL for traversing AST nodes. It is used to walk on the AST from node to node. This helps implement a code generator and a type checker.

To implement a tree walker, we define a subclass of `Checker`. `Checker` is a class provided by Yadriggy. Suppose that we deal with ASTs satisfying the following syntax:

```
syn = define_syntax do
  Binary <= { left: expr, right: expr }
  expr <= Binary | Number | Name
end
```

Then we can define a tree walker for these ASTs:

```
1  class SimpleChecker < Checker
2    rule(Number) do
3      'number'
4    end
5
6    rule(Name) do
7      'name'
8    end
9
10   rule(Binary) do
11     v1 = check(ast.left)
12     v2 = check(ast.right)
13     "#{v1}#{ast.op}#{v2}"
14   end
15 end
```

A rule such as `rule(Number) do` ... `end` in line 2 to 4 specifies the action when the tree walker visits a node of the given type such as `Number`. When a variable `tree` refers to the AST for the source code `a + b - 3`, then:

```
SimpleChecker.new.check(tree)
```

returns `"name+name-number"`.

The rules may return arbitrary values as their results. In the bodies of the rules, a special variable `ast` is available. Precisely, it is a method call without arguments to the `ast` method. `ast` refers to the AST node currently visited. Its type matches the type that the rule specifies for.

A rule can be defined for a user-defined type such as `expr`. If such a rule is included in the tree walker, the AST has to be checked by the syntax checker before running the tree walker so that AST nodes will be properly tagged. For example, the normal Ruby parser recognizes the following form of lambda expression as a method-call expression to the `lambda` method:

```
lambda {|x| x + 1 }
```

Thus the root node of the AST obtained by the `reify` method is a `Call` object representing a method call. If this is inconvenient, we can define a user-defined type `lambda_call` for the syntax checker and then use that type during the tree traversal to deal with lambda expressions separately from method-call expressions. The following rules define `lambda_call`:

```
lambda_name = { name: "lambda" }
lambda_call = Call + { receiver: nil, op: nil,
                       name: lambda_name,
                       args: nil, block_arg: nil,
                       block: Block }
```

It is defined as a method-call expression with a block argument but no receiver or arguments.

When the `check` method is invoked on the tree walker, it first attempts to find a rule for the user-defined type attached to the AST node passed to `check`. If the rule is not found, the `check` method attempts to find a rule for the most specific class for that AST node. It next attempts to find a rule for the superclass of the most specific class, and so forth. If no rule is finally found, the `check` method raises an exception.

We can also define a subclass of `SimpleChecker`, which is a subclass of the `Checker` class shown above. All the rules are inherited from the super class `SimpleChecker`. When a rule overrides the rule defined in the super class, the overridden rule can be invoked by calling the `proceed` method. Its parameter is the currently visited AST.

3.5 Applicability and Limitations

To access a foreign-language library, our approach migrates a code block written in a DSL embedded in the host language. The DSL borrows the host syntax although it interprets its code for a different semantics. Therefore, a host language with flexible syntax, such as Ruby, is appropriate to our approach. Static typing narrows down the design space of the DSL since the migrated code block has to be valid with respect to not only the syntax but also the types of the host language. Our approach, however, is also applicable to a host language with static typing and simple syntax, for example, the C language. The DSL embedded in that host language would be far different from the foreign language but it would be rather less confusing to the programmers. According to our observation, a DSL is confusing when the syntactically same expression is interpreted differently between the DSL and the foreign language. A very different DSL does not allow such a confusing expression although learning the DSL may require efforts.

The frameworks for our approach have to provide the `reify` method. The host language has to enable its implementation. It is crucial that `reify` extracts an AST for a lambda expression as well as a method/function body. In Python, extracting an AST for a function body by using an `@` decorator is a common technique. However, if `reify` only extracts an AST for a function body, the programmers have to write a separate function whenever they write the migrated code in the DSL. This is as inconvenient as writing an anonymous class in Java instead of a lambda expression. Furthermore, `reify` extracts an AST for an arbitrary lambda expression. The programmers do not need to annotate the lambda expression, for example, by an `@` decorator in Python, when they construct it.

The host language also has to enable to capture the runtime value of a free variable included in the migrated code block. Ruby's reflection ability allows `reify` to access the execution environment of a lambda expression. The `reify` method can obtain a variable name and its runtime value in that environment.

In our approach, the DSL interpreter has to report a syntax error when the migrated code is syntactically valid as the host-language code but not as the DSL code. A host language with highly flexible syntax enables a DSL with complex syntax. It is tedious to checking a syntax error in such a DSL. The frameworks for our approach should support the implementation of the syntax checker for such a DSL.

4 Case Studies

This section presents application examples of our framework, Yadriggy.

4.1 Syntax Checker

The syntax checker provided by Yadriggy is an application of Yadriggy itself. The DSL for the syntax checker could be implemented in the deep embedding style since most operators in Ruby are method calls and their method definitions are modifiable. Its implementation might not need our framework but it would be badly complicated. For example,

```
Binary <= { left: Number, right: Number }
```

This rule could be made available without our framework. `Binary` and `Number` are constants. They are class objects representing the classes `Binary` and `Number`, respectively. If we define the <= method for `Binary`, the expression above is executed as a method call to the <= method on `Binary` with a hash table as an argument:

```
Binary.<= ( { left: Number, right: Number } )
```

Since the rule is executed as a method call, we can properly implement the <= method so that the DSL will be implemented in the deep embedding style [21, 32, 34]. However, this technique does not work for user-defined types. When executing the following rule:

```
expr <= Binary | Number
```

it will throw an exception because `expr` is an undefined local variable. If we change the operator from <= to = and define

Listing 1. The DSL for the syntax checker

```
nil_value <= Reserved + { name: 'nil' }

ArrayLiteral <= { elements: [ array_elem ] }
array_elem <= Binary + {op: :*,left: array_elem,
           right: or_constraint } | or_constraint
Paren <= { expression: or_constraint }
constraint <= Const | IdentifierOrCall | Paren |
             ArrayLiteral | StringLiteral |
             SymbolLiteral | nil_value
or_constraint <= Binary +
              { op: :|, left: or_constraint,
                right: constraint } | constraint
HashLiteral <= {pairs: [ Label * or_constraint]}

operand <= Const | IdentifierOrCall |
            HashLiteral | nil_value
add_expr <= Binary + { op: :+, left: add_expr,
                       right: operand } | operand
expr <= Binary + { op: :|, left: expr,
                   right: add_expr } | add_expr

rule <= Binary + {left: Const| IdentifierOrCall,
                 op: :'=' | :'<=', right: expr }
Exprs <= { expressions: [ rule ] }
Parameters <= { params: [],
                optionals: [],
                rest_of_params: nil,
                params_after_rest: [],
                keywords: [],
                rest_of_keywords: nil,
                block_param: nil }
Block <= Parameters + { body: nil | rule | Exprs}
```

the `|` method on `Binary`, the rule will not throw an exception when it is executed. This solution unfortunately causes a problem for this rule:

```
Binary = { left: Number, right: Number }
```

This re-initializes the constant `Binary`. After the re-initialization, `Binary` is not a class name. We do not know a technique for addressing this problem.

Our framework enables us to give the DSL code a different semantics from Ruby; we can avoid the problem shown above. The `define_syntax` method provided by Yadriggy receives a lambda expression including the rules and it constructs a syntax checker without executing the body of that lambda expression. The AST constructed from the lambda expression matches the syntax rules in Listing 1. The syntax checker can be regarded as a foreign-language interpreter independent of Ruby. If so, the `define_syntax` method is a foreign language interface for migrating the code block written in the DSL for the syntax checker.

4.2 Python

We have implemented a foreign language interface to Python. When the code block is migrated to Python, copies of method definitions and variables are also migrated if the migrated code refers to them but they are outside the code. Those method definitions and variables are recognized by calling the `value` method on AST node objects obtained by the `reify` method.

The lower-level implementation of our interfacing system to Python relies on PyCall [22]. A reference to a Ruby object is copied by PyCall to Python as a remote reference, and vice versa. Primitive-type values and their arrays are passed by copying into Python. The Python code generated from the migrated code written in Ruby is also passed by PyCall to the Python interpreter.

The code migrated to Python is written in the DSL borrowing the syntax from Ruby. Our design goal for this DSL was to avoid that the programmers make a careless mistake due to minor differences between Ruby and Python, in particular, when the programmers are writing the migrated code based on Python tutorials. We assume that the programmers are familiar with Ruby but not Python.

When designing the DSL, we mapped Ruby's syntax to Python's language features in the following scheme. For each syntactic form in Python, if the same form is:

- available in Ruby
  - with the same semantics:
    Use *as is.*
  - with different semantics:
    Change Ruby's semantics so that this syntactic form will have the same semantics as in Python. For example, `True` in the DSL is mapped to `True` in Python although `True` does not mean a true value in normal Ruby. It is an undefined constant.

    Furthermore, if a different syntactic form in Ruby has that semantics in Python, use this form as well. For example, since `true` means a true value in Ruby, not only `True` but also `true` in the DSL are mapped to `True` in Python.
- unavailable in Ruby:
  - if a different syntactic form in Ruby has the same semantics:
    Use this form in Ruby. For example, a lambda expression in Ruby is syntactically different from one in Python but the Ruby-style lambda expression is used in the DSL.
  - otherwise:
    Use similar looking code in Ruby as the syntactic form in Python. We used this approach for list comprehension in Python.

A function definition in Python is expressed as a method definition in Ruby. It has to include a `return` statement although it can be omitted in normal Ruby since the result of the expression evaluated last is implicitly returned in Ruby. Our current implementation only supports required arguments; the default arguments, keyword arguments, or variable number of arguments are not supported for function definitions.

List comprehension is not available in Ruby. Hence we use a `for` statement enclosed by `[]` as a list comprehension. For example, `[for i in range(0, 3) do i end]` in the DSL is mapped to `[i for i in range(0, 3)]` in Python. Tuples are neither available in Ruby. `tuple()`, `tuple(1,)`, and `tuple(1, 2)` in the DSL are mapped to `()`, `(1,)`, and `(1, 2)` in Python, respectively.

The `with` statement is not available in Ruby, either. We express the `with` statement as follows:

```
with open('foo.txt') => f do
  print(f.read())
end
```

In Ruby, this is recognized as a method call to the `with` method with a hash argument `open('foo.txt') => f` and a block argument **do**...**end**. This is mapped to this Python code:

```
with open('foo.txt') as f:
  print(f.read())
```

To support this mapping, the method call to `with` in Ruby is tagged as `with_call` by the following syntax rule:

```
with_name = { name: "with" }
with_call = Call + { receiver: nil, op: nil,
                     name: with_name,
                     args: HashLiteral,
                     block_arg: nil, block: Block }
```

Only a hash literal is allowed for the argument.

The syntactic form for a keyword argument in a function call uses the = operator as in Python. It has a different meaning in Ruby. The syntactic form for a dictionary literal, such as `{x_data: 2.0}`, is differently interpreted between Ruby and Python. The DSL adopts Python's interpretation. Table 4 partly lists the syntactic mapping between the DSL and Python.

Listing 2 shows a Ruby program[8] using our framework for accessing the Python library, matplotlib. The `draw_pie` method from line 1 to 8 and the code block in line 15 and 16 are migrated to Python. The values of `labels` and `sizes` are also copied to Python. The Python program corresponding to Listing 2 is shown in Listing 3. Only line 15 in Listing 2 and line 14 in Listing 3 are fundamentally different.

5 Related Work

Our framework Yadriggy is similar to Lisp macro systems [7, 36, 38]. However, Yadriggy's `reify` method extracts an AST for the given lambda expression or a method object whereas a macro system extracts an AST for the macro argument. A macro system cannot extract an AST from a first-class value like a lambda expression. A macro function can deal with the extracted AST only at macro-expansion time and it has to return a modified AST, which will be executed instead of the original one. Furthermore, the `reify` method deals with the complex syntax of Ruby whereas a Lisp macro system deals with simple S-expressions. To deal with the complex syntax, Yadriggy provides a syntax checker.

A number of macro-like systems for programming languages with more complex syntax have been proposed. Squid [29] is a code rewriting system for Scala. It combines code rewriting and staged computing for user-defined optimization. We could tweak Squid to use for developing a foreign language interface but we would have to write a number of rewriting rules since Squid is not designed for that purpose. The rewriting rules would transform the code to construct an AST in the deep embedding style.

Table 4. The DSL for the foreign language interface to Python

DSL	Python	example
true	True	b = true
false	False	b = false
True	True	b = True
False	False	b = False
nil	None	x = nil
None	None	x = None
!	not	!true
&&	and	x > 3 && y > 2
||	or	x > 3 || y > 2
.in	in x	x.in [1, 2, 3]
		x.in([1, 2, 3])
.not_in	not in	x .not_in [1, 2, 3]
.idiv	//	x .idiv 3
i..j	range(i,j)	for i in 0..n do sum += i end
range(i,j)	range(i,j)	for i in range(0, n) do sum += i end
? :	if else	x > 0 ? x + 1 : -x + 1

Listing 2. Use of matplotlib from Ruby

```
1  def draw_pie(labels, sizes, explode)
2    fig1, ax1 = plt.subplots()
3    ax1.pie(sizes, explode=explode, labels=labels,
4            autopct='%1.1f%%', shadow=True,
5            startangle=90)
6    ax1.axis('equal')
7    plt.show()
8  end
9
10 def run()
11   import_py('matplotlib.pyplot').as(:plt)
12   labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
13   sizes = [15, 30, 45, 10]
14   run_python do
15     ex = tuple(0, 0.1, 0, 0)
16     draw_pie(labels, sizes, ex)
17   end
18 end
19
20 run
```

[8]This was taken from the matplotlib tutorial: https://matplotlib.org/gallery/pie_and_polar_charts/pie_features.html

Listing 3. The corresponding Python program

```
import matplotlib.pyplot as plt

def draw_pie(labels, sizes, explode):
  fig1, ax1 = plt.subplots()
  ax1.pie(sizes, explode=explode, labels=labels,
          autopct='%1.1f%%', shadow=True,
          startangle=90)
  ax1.axis('equal')
  plt.show()

def run():
  labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
  sizes = [15, 30, 45, 10]
  ex = (0, 0.1, 0, 0)
  draw_pie(labels, sizes, ex)

run()
```

SugarJ [9] extends Java to enable a library that provides a user-defined syntax. The extended syntax is transformed into the syntax of regular Java. The library defines this transformation by several rewriting rules for concrete syntax. It does not use ASTs. TSL Wyvern [26] allows user-defined syntax only within a type-directed scope. ProteaJ2 [19] also allows user-defined syntax in a type-directed scheme. For execution, the new syntax is not transformed into the host-language syntax; instead, it is treated as a call to the function associated with that syntax. Recaf [3] allows a new syntactic form starting with a user-defined keyword. The new form is also treated as a call on its semantic object. To run in unmodified Ruby, Yadriggy does not enable syntax extensions by the users. It lets the users reuse existing syntax in Ruby by rather limiting available forms of syntax. A syntax checker is provided for this purpose.

There have been several systems for constructing an IR (Intermediate Representation) for the given code block. Lancet [33] extracts the IR for Java bytecode during runtime from the Graal VM [27, 28]. It uses their IR for customizing JIT compilation in Java. Their IR is not designed for changing the semantics or the interpretation of the original code block although Yadriggy's ASTs are for changing the semantics so that the code block can be regarded as foreign language code. Numba [23] also extracts the IR for Python by decompiling Python bytecode at runtime. It also uses the IR for JIT compilation by LLVM.

Like Yadriggy, our previous work Bytespresso [6] constructs the AST for a lambda expression. This system constructs the AST by decompiling Java bytecode during runtime and uses it for migrating Java code to GPU. However, this system is for Java, a statically typed language, and thus does not radically change the semantics or interpretation of the Java code. It would be difficult to radically change the semantics while keeping the code correctly typed. On the other hand, Yadriggy exploits the fact that Ruby is dynamically typed and Ruby programs do not include explicit type declarations. Yadriggy provides a syntax checker for mitigating the complexity of the ASTs.

Deep embedding is a technique widely used for constructing an AST for a method-call chain to a fluent interface [11, 13, 14, 40]. Scala LMS [32] exploits Scala's type system to easily construct an AST for deeply embedded code and enable staged computing. Scala-Virtualized [24, 31] gives more natural presentation to that embedded code. Unlike these deep embedding systems, Yadriggy directly constructs an AST from the source code. The user programmers do not have to write client code in the deep embedding style. A drawback of deep embedding against quotation-based macros is also discussed in [25]. Besides, Yadriggy does not require that the host language is statically typed or supports an advanced type system like Scala's.

PyHyp [2] is a hybrid interpreter of PHP and Python. It can run a PHP program including Python code and vice versa. Like our approach, PHP and Python code can call a function and also access a variable in the other language. However, the PyHyp programmers have to use a dedicated source-code editor *Eco*, which performs source-level transformation and generates PHP code where Python code is embedded as a string literal. Our approach allows us to use normal Ruby and Python interpreters and their normal editors since it provides an embedded DSL for accessing Python from Ruby.

Although our work mainly focuses on foreign language interface, the idea of migrating a code block to somewhere different has been actively studied in the context of distributed computing. Remote procedure calls (RPCs) are prevalent communication abstracts based on function calls but they imply non-trivial performance penalties. Remote batch invocation (RBI) [17] is one of the solutions. It is a mechanism for sending code to a remote site as a batch and executing it in a single round-trip. The original RBI system was proposed for a single-language environment based on Java. Yadriggy can be used to implement RBI in a heterogeneous environment involving different languages.

6 Conclusion

This paper presented that the interface based on code migration is an alternative to traditional foreign function interfaces. The contributions of this paper are the following. First, it showed that a function call is not always an appropriate abstraction for accessing a library written in a foreign language, in particular, a modern programming language such as Python. Another contribution is that the paper proposed a new framework *Yadriggy* for implementing a foreign language interface based on code migration. The framework works in Ruby. It provides a facility to extract an AST from a code block given as a lambda expression or a method object. It also provides a syntax checker for processing a code block written in a language with complex syntax.

Listing 4. The syntax rules for Ruby

```
expr <= Name | Number | Super | Binary | Unary |
       SymbolLiteral | ConstPathRef |
       StringLiteral | StringInterpolation |
       ArrayLiteral | Paren | Call | ArrayRef |
       HashLiteral | Return | ForLoop | Loop |
       Conditional | Break | Lambda | BeginEnd |
       Def | ModuleDef
exprs <= Exprs | expr
Name <= { name: String }
Number <= { value: Numeric }
Super <= {}
Identifier <= Name
SymbolLiteral <= { name: String }
VariableCall <= Name
InstanceVariable <= Name
GlobalVariable <= Name
Label <= Name
Reserved <= Name
Const <= Name
Binary <= { left: expr, op: Symbol, right: expr}
ArrayRef <= { array: expr, indexes: [ expr ] }
ArrayRefField <= ArrayRef
Assign <= { left: [expr] | expr, op: Symbol,
            right: [expr] | expr }
Dots <= Binary
Unary <= { op: Symbol, operand: expr }
ConstPathRef <= { scope: (ConstPathRef | Const),
                  name: Const }
ConstPathField <= ConstPathRef
StringLiteral <= { value: String }
StringInterpolation <= { contents: [ exprs ] }
ArrayLiteral <= { elements: [ expr ] }
Paren <= { expression: expr }
HashLiteral <= { pairs: [ expr * expr ] }
Return <= { values: [ expr ] }
ForLoop <= { vars: [ Identifier ], set: expr,
             body: exprs }
Loop <= { op: Symbol, cond: expr, body: exprs }
Conditional <= { op: Symbol, cond: expr,
          then: exprs, all_elsif: [expr * exprs],
          else: (exprs) }
Parameters <= { params: [ Identifier ],
               optionals: [ Identifier * expr ],
               rest_of_params: (Identifier),
               params_after_rest: [ Identifier ],
               keywords: [ Label * expr ],
               rest_of_keywords: (Identifier),
               block_param: (Identifier) }
Block <= Parameters + { body: exprs }
Lambda <= Block
Call <= { receiver: (expr), op: (Symbol),
         name: (Identifier), args: [ expr ],
         block_arg: (expr), block: (Block) }
Command <= Call
Exprs <= { expressions: [ expr ] }
Rescue <= { types: [ Const | ConstPathRef ],
          parameter: (Identifier),
          body: (exprs), nested_rescue: (Rescue),
          else: (exprs), ensure: (exprs) }
BeginEnd <= { body: exprs, rescue: (Rescue) }
Def <= Parameters +
       { singular: (expr), name: Identifier,
          body: exprs, rescue: (Rescue) }
ModuleDef <= { name: Const | ConstPathRef,
               body: exprs, rescue: (Rescue) }
ClassDef <= ModuleDef +
           { superclass: (Const | ConstPathRef) }
SingularClassDef <= { name: expr, body: exprs,
                      rescue: (Rescue) }
Program <= { elements: exprs }
```

A The AST Obtained by `reify`

The specification of the ASTs obtained by the `reify` method is described by the syntax rules in Listing 4.

Acknowledgments

This work was partly supported by DFG/JST SPPEXA ExaStencils project and JSPS KAKENHI Grant Number JP15H02682.

References

[1] Adam Paszke and Sam Gross and Soumith Chintala and Gregory Chanan. 2016. PyTorch. http://pytorch.org/.

[2] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2016. Fine-grained Language Composition: A Case Study. In *30th European Conf. on Object-Oriented Programming (ECOOP 2016)*, Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 3:1–3:27.

[3] Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. 2016. Recaf: Java Dialects As Libraries. In *Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Generative Programming: Concepts and Experiences (GPCE 2016)*. ACM, 2–13.

[4] Shigeru Chiba. 1995. A Metaobject Protocol for C++. In *Proc. of ACM OOPSLA*. ACM, 285–299.

[5] Shigeru Chiba. 2019. Artifact - Foreign Language Interfaces by Code Migration. https://doi.org/10.5281/zenodo.3387514

[6] Shigeru Chiba, YungYu Zhuang, and Maximilian Scherr. 2016. Deeply Reifying Running Code for Constructing a Domain-Specific Language. In *Proc. of the 13th Int'l Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. ACM, Article 1, 12 pages.

[7] William Clinger and Jonathan Rees. 1991. *Revised^4 Report on the Algorithmic Language Scheme.*

[8] DoRuby. 2018. Tensorflow MNIST tutorial by Ruby and PyCall. https://doruby.jp/users/tsuji/entries/.

[9] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proc. of ACM OOPSLA*. ACM, 391–406.

[10] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*. ACM, 111–122.

[11] Martin Fowler. 2005. FluentInterface. https://www.martinfowler.com/bliki/FluentInterface.html.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns*. Addison-Wesley.

[13] Yossi Gil and Tomer Levy. 2016. Formal Language Recognition with the Java Type Checker. In *30th European Conf. on Object-Oriented Programming (ECOOP 2016) (LIPIcs)*, Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 10:1–10:27.

[14] Yossi Gil and Ori Roth. 2019. Fling - A Fluent API Generator. In *33rd European Conf. on Object-Oriented Programming (ECOOP 2019) (LIPIcs)*, Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 13:1–13:25.

[15] GitHub, Inc. 2019. Most popular languages on GitHub as of February 18, 2019. https://github.com/oprogramador/github-languages#all.

[16] Google Brain. 2015. TensorFlow. https://www.tensorflow.org/.

[17] Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. 2009. Remote Batch Invocation for Compositional Object Services. In *Proc. of the 23rd European Conf. on ECOOP 2009 — Object-Oriented Programming*. Springer-Verlag, 595–617.

[18] Kazuhiro Ichikawa and Shigeru Chiba. 2014. Composable User-defined Operators That Can Express User-defined Literals. In *Proc. of the 13th Int'l Conf. on Modularity (MODULARITY '14)*. ACM, 13–24.

[19] Kazuhiro Ichikawa and Shigeru Chiba. 2017. User-Defined Operators Including Name Binding for New Language Constructs. *The Art, Science, and Engineering of Programming* 1, 2 (2017), Article 15.

[20] John D. Hunter. 2003. Matplotlib. https://matplotlib.org.

[21] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: Concealing the Deep Embedding of DSLs. In *Proc. of the 2014 Int'l Conf. on Generative Programming: Concepts and Experiences (GPCE 2014)*. ACM, 73–82.

[22] Kenta Murata. 2016. PyCall: Calling Python functions from the Ruby language. https://github.com/mrkn/pycall.rb.

[23] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-based Python JIT Compiler. In *Proc. of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, 7:1–7:6.

[24] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized. In *Proc. of Workshop on Partial Evaluation and Program Manipulation (PEPM '12)*. ACM, 117–120.

[25] Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-specific Languages. In *Proc. of Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, 25–36.

[26] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *Proc. of the 28th European Conf. on ECOOP 2014 — Object-Oriented Programming (LNCS 8586)*. Springer-Verlag New York, Inc., 105–130.

[27] Oracle. 2012. OpenJDK: Graal project. http://openjdk.java.net/projects/graal.

[28] Oracle Corp. 2019. GraalVM: Run Programs Faster Anywhere. https://www.graalvm.org.

[29] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *Proc. of the 16th ACM SIGPLAN Int'l Conf. on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, 131–145.

[30] Qiita. 2018. Tips for PyCall Ruby. https://qiita.com/stat/items/a383451e7f824e4c9627.

[31] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* 25, 1 (2012), 165–207.

[32] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. of the Ninth Int'l Conf. on Generative Programming and Component Engineering (GPCE '10)*. ACM, 127–136.

[33] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical Precision JIT Compilers. In *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '14)*. ACM, 41–52.

[34] Maximilian Scherr and Shigeru Chiba. 2015. Almost First-Class Language Embedding: Taming Staged Embedded DSLs. In *Proc. of Int'l Conf. on Generative Programming: Concepts and Experiences (GPCE 2015)*. 21–30.

[35] B. C. Smith. 1984. Reflection and Semantics in Lisp. In *Proc. of ACM Symp. on Principles of Programming Languages*. ACM, 23–35.

[36] G. Steele. 1990. *Common Lisp: The Language* (2nd ed.). Digital Press.

[37] Steven G. Johnson. 2013. PyCall: Package to call Python functions from the Julia language. https://github.com/JuliaPy/PyCall.jl.

[38] Daniel Weise and Roger Crew. 1993. Programmable Syntax Macros. In *Proc. of Conf. on Programming Language Design and Implementation*, Vol. 28, no. 6. ACM SIGPLAN Notices, 156–165.

[39] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[40] Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. 2019. Generating a fluent API with syntax checking from an LR grammar. In *Proc. of ACM OOPSLA*. ACM.