View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Konoha: Implementing a Static Scripting Language with Dynamic Behaviors

Kimio Kuramitsu
Yokohama National University, JAPAN
kimio@ynu.ac.jp

Abstract

This paper presents the design of Konoha, a statically typed object-oriented scripting language. Konoha is modeled to have the same or similar scripting experience with dynamic languages, by emulating major dynamic behaviors, such as duck typing and eval function. At the same time, its ”run anytime” complier enables execution of incomplete programs without the compilation stop by the static type checker. Konoha was written in C from scratch, and is available as open-source software. We will show scripting experiences, as well as better performance.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Object-oriented scripting languages

Keywords scripting, language design, dynamic languages

1. Introduction

Dynamic languages such as Python, Ruby and JavaScript have gained broad acceptance in industry, especially in web application development. One of the major reasons lies in their rapid software update cycle in both the development and deployment phase. In particular, dynamic behaviors fit well with the rapid catch-up with changed user demands.

On the other hand, the lack of static types in existing scripting languages has received growing attention, as their scripts are larger and more complex especially in web application development. Since static languages such as C++ and Java have a rich history that enables a program to be robust and efficient, many language developers are motivated to make an attempt to add static typing features into existing dynamic languages. The development of Diamondback Ruby[4] and Restricted Python[1] are two interesting attempts to the integration of static typing features into Ruby and Python, respectively.

We take a different approach to integrating dynamic language features into a statically typed object-oriented scripting language — our newly designed language named Konoha. This paper present the design and implementation status of Konoha.

In this paper, we aim at language design for providing the same or similar scripting experience with Perl and Ruby. This experience is in general regarded as attributes enabling “ease of programming”. We argue that these attributes are not exclusive to dynamic typing and can be modeled in a static language.

In order to model dynamic language features, we focus on four major dynamic behaviors that are commonly recognized in existing scripting languages. These are: dynamic alteration of object behaviors, duck typing, handling of strings as executable programs, and non-check cycle of program execution. In Konoha, we have implemented these programming features with some static-typing considerations:

- *Growing class* allows the runtime alteration of object behaviors while keeping type safety.
- *Type inference* allows the use of non-declared variables in an integrated manner with statically typed variables.
- *Dynamic type* allows the duck typing style of programming.
- *“Run anytime” compiler* allows the execution of partially written programs by automated replacement of error code with code throwing a runtime exception.

Konoha is developing as open source software. The reader can download the latest version from the following sites:

```
http://code.google.com/p/konoha
http://konoha.sourceforge.jp/
```

The paper is organized as follows. Section 2 discusses the programming experiences in dynamic languages. Section 3 presents our language design, implemented in Konoha. Section 4 reports implementation details with some earlier

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop on Self-sustaining Systems (S3) 2010
September 27–28, 2010, The University of Tokyo, Japan
Copyright © 2010 ACM 978-1-4503-0491-7/10/09...$10.00

performance results. Section 5 reviews related work. Section 6 concludes the paper.

2. Dynamic Behaviors in Dynamic Languages

Over decades, dynamic languages have gained popularity with a very large fraction of application developers, mainly because scripting languages are considered to offer some "*ease of programming*" attributes. Although this reputation seems too subjective to discuss, we would like to start by rethinking what enables "ease of programming" in dynamic languages.

2.1 "Ease of Programming" Attributes in Scripting Languages

Our motivation is the language design of a statically typed scripting language sharing the same, or very similar, programming experience with existing dynamic languages such as Python and Ruby. The programming experience that we focus on in this paper is unclearly recognized as "ease of programming" attributes that are enabled by dynamic typing. We argue that these attributes are not exclusive to dynamic languages and can be modeled on top of a static language.

To begin with, we focus on the following four dynamic behaviors that commonly appear in existing dynamic languages.

- Runtime alteration of objects, including field addition, method overriding, and missing method
- (Dynamic) structural typing (as known as *duck typing*)
- Handling of strings as executable programs (via `eval()`)
- Execution of partially written programs

We consider that these four dynamic features are strongly related to programming functions that enables "ease of programming" in a scripting language. As shown in Table 1, these features are well shared in typical scripting languages, such as Perl, Python, Ruby, Lua, and Groovy.

In the remainder of this section, we would like to review each of these programming features with some remarks on design considerations for static typing. For readability, we write all sample sources with a Java-based grammar.

2.2 Runtime Alteration of Objects

Most dynamic languages provide a means for altering a member of existing objects at runtime. The class design in programming can be flexible, by allowing objects to be altered differently depending on branched program execution paths. The usefulness of this feature is known in several practical programming scenarios, especially in case of handling semi-structured data, such as XML and JSON.

Here is an example of *field addition*, commonly appearing in dynamic language programming. The `age` field that is not declared in the `Person` class is *implicitly* added to a `Person` object by the field accessor.

```
class Person {
  String name;
  Person(String name) {
    this.name = name;
  }
}

Person p = new Person("naruto");
p.age = 17; // implicit addition (not type error)
```

In addition to the field addition, *method overriding* at runtime and handling *missing methods* are widely used in some of dynamic languages.

Dynamic languages, on the other hand, provide no protection for object alteration both at compilation time and run time. As can be easily imagined, the alteration of method types is likely to cause significant inconsistency, which would be usually observed as unexpected runtime errors. In a static language, type safety must be preserved after the alteration of object members.

2.3 Type Declaration

The absence of explicit type declaration is the most remarkable part of programming in dynamic languages. Non-static constraint between types and variables allows programmers to bind any types of values to arbitrary existing variables. This allows the reuse of variables for different typed values, although it is hard to distinguish from careless mistakes.

```
s = "string";
s = 1;     // bugs? reuse ??
```

Local type inference is known to provide implicit type declaration, which is nearly the same style in appearance and facilitates an agile style of programming as well. In addition, the static analysis of JavaScript programs reported in [10] indicates that the reuse of variables across different types is not a main part of dynamic language programming.

It is important to note that dynamic typing is by nature based on the *structural* type system. The type equivalence is different from that in the nominative type system. To illustrate the difference, suppose the following two unrelated classes: `Person` and `Dog`.

```
class Person {
  String name;
  Person(String name) { .. }
}
class Dog {
  String name;
  Dog(String name) {.. } ;
}
void hello(w) {
  print w.name;
}

Person p = new Person("Naruto");
Dog d = new Dog("Hachi");
hello(p)
hello(w)
```

Behaviors	Perl	Python	Ruby	JavaScript	Lua	Groovy	Java	Scala	Konoha
typing	dynamic	dynamic	dynamic	dynamic	dynamic	gradual	static	static	static
field addition	+	+	+	+	+	+			+
method addition		+	+	+	+	+			+
method rewriting		+	+	+	+	+			+
missing method		+	+		+	+			
duck typing	+	+	+	+	+	+		static, explicit	`Any`
eval	+	+	+	+	+	+			+
non-check execution	+	+	+	+	+	+			+

Table 1. Comparison of language supports for dynamic behaviors

The structural type system tries to check type equivalence by the structure of object members. In a statically typed language, an abstract class or an interface class to type the variable `w` in `hello()` is necessary, while in a dynamic language it is not necessary because the type of the variable `w` is checked at runtime when `w.name` is called. As we described above, the possibility of object alterations can create a variety of similar objects at runtime — probably resulting in more difficult declaration of interface classes. Because dynamic languages requires no declared types for object class, as a result they better deal with the abstraction of unrelated objects. This feature is called *duck typing* among dynamic language programmers[1]. Although duck typing is nearly a *dynamic structural subtyping*, we use the term duck typing in this paper.

2.4 Handling of Strings as Executable Programs

Many dynamic languages allow us to transform a string form of source code into executable program at runtime. Moreover, the transformed executable can be executed with bindings of variables and other symbols in the context of transformation. This mechanism is called *eval*, and is typically provided through the `eval()` function.

```
eval("a = f(1)")
print a;
```

The `eval()` function is not new in the programming language literature, but it provides significant advantages of program delivery and software updates over static languages, whose code generation is usually decoupled from code execution environments. The popularity of JavaScript on the Web is considered to a typical case showing these advantages.

Used together with dynamic alteration of object behaviors, runtime evaluation of program becomes a more crucial part in modern software updates cycle. However, a traditional eval mechanism is not so perfect to fit with the Web age. Apparently, the `eval()` function in an existing scripting language has several difficulties in security and program safety. The `eval()` function is increasingly necessary and needs to be improved by static typing in terms of software safety.

[1] *—If it waddles like a duck, and quacks like a duck, it's a duck!*

2.5 Non-Check Cycle of Program Execution

Dynamic languages were originally designed to run on an interpreter. Although many of today's modern implementations adopted a compiler-based code generation for virtual machine's bytecode or native CPU instructions, they still run programs in an interpreted and interactive style of program execution. That is, programs can be run without explicit compiler's breaks, because the integrated compiler cannot check type errors.

The advantage of fewer compiler checks lies in its rapid development cycle from coding to testing. Because code generation is rarely aborted at compilation time, programmers can almost always execute the interesting parts of their incomplete programs, and understand program behaviors at a very earlier stage.

We argue that this "non-check" cycle brings a lot of impacts on "ease of programming" in dynamic languages, although it might deny benefits of static type checking.

3. Static Scripting with Konoha

We have been developing a static scripting language, named Konoha[2], since 2006. The design goal of Konoha is, say, it "looks like Java, runs like Python". Indeed, most of Konoha's grammar is clone of Java's, but some of the remainder are properly simplified to program.

3.1 Konoha Basics

Konoha is a static scripting language, written in C from scratch. As we are still improving Konoha every day, the latest version of Konoha 0.7 has the following features:

- Imperative language (C/C++, Java-style grammar)
- "pure" object-oriented programming ("everything is an object", a name-based class system, single inheritance, and generics)
- Python-style interactive shell
- open source, running on various platforms, including Linux, and MaxOS X, Windows, Android OS.

[2] Konoha means leaves of tree in Japanese. We named it after the Hidden Village of Konoha on Naruto.

The language design of Konoha is chiefly influenced by Java. Most of the control statements and all the operators in Java are imported to Konoha without any semantic changes. Basic types, such as `Object`, `Boolean`, `Integer`, and `Float`[3] and basic classes such as `System` and `InputStream` are named after Java class libraries. Thus, the readers may read all the Konoha's sample sources as if it is Java.

Here is the first Konoha sample for a `Person` class, written as compatible with Java at the source level.

```
class Person {
  // fields
  String name;
  int age;
  // constructor
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  // methods
  String getName() { return this.name; }
}
```

In addition, Konoha provides us with a Python-style interactive shell, where >>> is a command prompt to *eval* an inputted statement or expression, and the resulting evaluated value is printed out on the next line.

```
$ konoha
Konoha 0.7(aomori) source (rev:1751, Jul 29 2010)
[GCC 4.2.1 (Apple Inc. build 5659)] on macosx
Options: rcgc thread used_memory:353 kb

>>> int n = 1;
>>> n + 1
2
```

The above `Person` class can be instantiated on a Konoha shell. First, we declare a new class `Person`, and then use the new operator to construct a new object of `Person`.

```
>>> class Person {
  String name;
  int age;
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  String getName() { return this.name; }
}
>>> Person p = new Person("Naruto Uzumaki", 18)
>>> p.getName()
"Naruto Uzumaki"
```

Konoha is open source software and available at the following site.

http://codes.google.com/p/konoha
http://konoha.sourceforge.jp/

[3] For the compatibility with Java, Konoha defines some aliasing rules for primitive types(e.g., boolean as Boolean, int as Integer).

3.2 Dynamic Alteration of Objects

As we described in Section 2.2, the runtime alteration of object behaviors is a crucial part of dynamic languages. At the same time it is likely to destroy type safety. As Konoha must be type safe, we have to avoid some kinds of alterations that violate type safety.

Here are three kinds that can be safe in the alteration.

- The addition of new methods to existing classes
- The addition of new fields to existing classes
- The modification of existing methods with no alteration of function interfaces

3.2.1 Addition of a Method

Konoha allows us to define an additional method outside of its class declaration. Here is a new method `isChild()` that is added to the `Person` class. Due to the object privacy, only public field accessors such as `this.age` are allowed.

```
boolean Person.isChild() {
  return (this.age <= 21);
}
```

The addition of a new method affects all instantiated objects. The following demonstrates the addition of the method `Person.isChild()` that is undefined in the `Person` before the addition.

```
>>> p = new Person("Naruto Uzumaki", 18);
>>> p.isChild()
 - [(eval):1]:(error) undefined method:
      Person.isChild

>>> boolean Person.isChild() {
      return (this.age <= 21);
    }
>>> p.isChild()
true
```

Adding fields can be done in the same fashion. To start, Konoha provides programming transparency between getter/setter methods and field accessors. That is, `p.firstName` is simply a syntax sugar of `p.getFirstName()`.

```
>>> String Person.getFirstName() {
  return this.name.split()[0];
}
>>> p.name
"Naruto Uzumaki"
>>> p.firstName
"Naruto"
```

A small trick is required in adding an undeclared field. First, we suppose there exists a dictionary map that is able to store any values with associated string keys. Although this map can be placed on a global variable in theory, we assume that for convenient `setMetaData()`/`getMetaData()` are defined to wrap the access to the map. Using these functions, we emulate a virtual field accessor whose field does not exist in the class declaration.

Here is an example of explicitly adding the `mother` field to the `Person` class.

```
void Person.setMother(Person p) {
  void setMetaData(this, "mother", p);
}
Person Person.getMother() {
  return (Person)getMetaData(this, "mother");
}

>>> p.mother = new Person("Kushina", null);
>>> p.mother
Person{name: "Kushina", age: 0}
```

3.2.2 Modification of a Method

In Konoha, the redefinition of compiled methods means the modification of its compiled body. This is said the *runtime method overriding* on the same object. The control of overriding is unnecessary, because the affection of modified code is propagated to all instantiated objects as a replacement of old code.

```
>>> p = new Person("Naruto Uzumaki", 18);
>>> p.isChild()
true
>>> Boolean Person.isChild() {
      return this.age <= 17;
}
>>> p.isChild()
false
```

Konoha disallows the update for different types of method interface. If a method interface is altered, compiled code calling the modified method causes type errors. That is, we have to keep the equivalence of method interface in the method overriding — which is easy to check because all defined methods in Konoha are well typed.

3.2.3 Deletion of a Method

Deletion of fields or methods has a disruptive influence on the type safety of already compiled code. Meanwhile, the behaviors of deleted method calls are not common in dynamic languages. For example, Ruby calls *missing methods*, others just raises a no such method exception at runtime. The problem is that static languages usually have no dynamic checking supports in method calls, and additionally protect the redefinition of deleted method as different method types.

Konoha allows abstract method call due to an agile style of development, which is supposed to return just a default value with some warning message. Deletion of a method can be done the redefinition of its method as an abstract method to preserve types of method parameters and return value.

```
// redefinition as abstract method
>>> boolean Person.isChild();

// abstract methods are defined
>>> p.isChild()
false
```

Note that Konoha has adapted the null object pattern, where `null` is defined as default values of each types. In the above, `isChild()` method returns `null` of `boolean` (`false`).

3.3 Local Type Inference

As Konoha is a static language with Java-style grammar, types can be declared in a form that variable names follow its type.

```
>>> String s1;
>>> s1
null
```

Basically, variables are available only if their types are statically declared. However, this rule is often cumbersome, especially in an agile style of development, because some types are apparent from contexts. We allow one exception in the assignment of undefined variables by using a simple local *type inference* of the left-hand variable from its right-hand expression.

```
>>> s2
 - [(eval):1]:(error) undefined variable: s2
>>> s2 = "naruto"
"naruto"
>>> typeof(s2)
konoha.String

>>> s2=1
 - [(eval):1]:(type error) not numeric: String
```

In addition, Konoha has the *generics* support, such as `Array<String>` and `Iterator<String>`, which enables more precise type inference for elements of the given collection class. In the following example, variable `s` is inferred as `String` from `Iterator<String>`.

```
foreach(s in ["a", "b", "c"]) {
  print s;  // s is inferred as String
}
```

The local type inference seems similar to dynamic typing for variables, but it is significantly different in terms of variable scope and duck typing.

3.3.1 Scope of Local Variables

Many dynamic languages have adopted a function-wide scope because of no explicit type declaration. Thus, the following local variable `s` is accessible at any blocks in the whole function.

```
String typeofnum(int n) {
  if(n % 2 == 0) {
    s = "even";
  }
  else {
    s = "odd";
  }
  return s;
}
```

Konoha, on the other hand, have adopted the block-wide scope that is very common in most static languages. That is, the local variable `s` above is regarded as *undefined* at the last `return` statement. This might decrease the agility of programming to some extent. However, the consistency of scope rule leads to less misunderstanding. In addition, the

block-wide scope allows the safe reuse of differently typed variables, as follows:

```
{
  a = [];  // Array
  {
    int a = 1; // re-declaration as int
  }
  print a[0] // a can be access as a
}
```

3.3.2 `Any` type for Dynamic Type Checker

While the local type inference allows the omission of explicit type declarations like dynamic typing, the types of variables are decided at compilation time, and variables do not accept values of another type. In [10], the analysis of JavaScript indicated that almost all variables are used as if a single static type. In many cases, we consider that static typing by type inferences has no crucial impacts on "ease of programming". However, as we discussed in Section 2.3, dynamic typing is useful for the abstraction of structurally similar objects.

In Konoha, we define `Any` type to incorporate dynamic typing into a part of static type system. As its name implies, `Any` type accepts arbitrary type at compilation time. Instead, a dynamic type checker is inserted before the access to values of `Any` type. The dynamic type checker raises a type error at runtime if the operation is undefined.

```
>>> Any p = new Person("sasuke", 18);
>>> p.getName()
 "sasuke"

>>> p.getFriends()
 ** NoSuchMethodException: Person.getFriends()
```

3.4 "Run anytime" Compilation

A static type checker provides a strong means for assuring some parts of program correctness. However, writing a correct program takes long time, meaning the program can often not be executed when a programmer wants to test their unfinished program. As we described in Section 2.5, this becomes an considerable obstacle for doing an agile style of development in static languages.

Although Konoha's compilation process includes static type checking, it is also designed to allow the same development cycle with existing dynamic languages. Detected errors at compilation time are all reported to users, as other static languages do. At the same time, detected errors are replaced with some safely executable code instead of canceling the entire compilation process. Thus, Konoha can run a program anytime as scripting languages do.

Let us suppose that `readLine()` must return `String` value. Therefore, the following function includes a type error at the first block of the `if` statement.

```
int newSerialNum (int n) {
  if(n == 0) {
    InputStream in = new ("serial.txt");
    n = in.readLine(); // a type error
    in.close();
  }
  return n + 1;
}
```

In Konoha, the detected type error is replaced with code throwing an runtime exception `Source!!`. The following is a rewritten version of the above. Note that `!!` is a shorthand for `-Exception`.

```
int newSerialNum (int n) {
  if(n == 0) {
    throw new Source!!();
  }
  return n + 1;
}
```

We consider that a program catching a `SourceException` is *not* regular.

The throwing code would be inserted at the top of the branched block where the occurrence of a type error is inevitable. We do this to avoid the partial execution of a path that leads to an error. On the other hand, execution paths where the occurrence of a type error is not deterministic are compiled as usual. Accordingly, we can run the function `newSerialNum ()` correctly if the error path is not hit.

```
>>> newSerialNum(1)
2
>>> newSerialNum(0)
 ** Source: in Script.newSerialNum
```

4. Implementation of Konoha

We have incrementally improved the design and the implementation of Konoha, since we released the first version as konoha 0.1 in 2008. Here we describe the implementation based on the latest version (konoha 0.7), which is not officially released yet.

4.1 Konoha System and `eval()`

As with scripting languages, Konoha is designed to have integrated code generation with its code execution engine. The `eval()` function is an interface of the integrated compiler and execution process, chiefly including:

- a parser that tokenizes scripts and then constructs ASTs,
- a type checker that types each of AST nodes (using type inference),
- a code generator (including a simple optimization process),
- a code execution engine, called Konoha virtual machine

The `eval()` function accepts three forms of scripts, namely, declarations (i.e, classes and methods), expressions (i.e., operators and function calls) and statements (i.e., `if`, `while`, etc). The eval of declarations just adds the structure information of classes and the compiled code of methods into the class table. The eval() of expressions and statements

```
>>> Any n = 1   // global variable, needs dynamic type checking
>>> n++
L1(1): MOVx sfp[3] sfx[0]+0    // move global variable to stack[3]
L2(1): TYPECHK sfp[3] Int     // typecheck because of stack[3] is typed Any
L3(1): UNBOX sfp[3]          // unbox
L4(1): iINC sfp[3]             // increment sfp[3]
L5(1): BOX sfp[3] Int          // box sfp[3] as Int
L6(1): XIMOV sfx[0]+0 sfp[3]   // move an object stack[3] to the global
```

Figure 1. Compiled virtual machine code that includes dynamic type checking

Table 2. Micro benchmark: score indicates million operations per second. (Larger is faster.) Java VM was run without the JIT compilation.

Microbench	g++4.2 (O0)	Java6 (VM)	konoha 0.7	Lua 5.1	ruby1.9.1	python2.6	Perl 5.8
SimpleLoop	299	66.9	209	53.2	10.08	11.4	13.0
LocalVariable	> 1000	386	526	32.1	45.2	65.8	25.6
GlobalVariable	> 1000	141	239	25.8	29.1	23.2	27.0
StringAssignment	30.7	167	280	22.3	4.13	58.6	22.7
IntegerOperation	> 1000	531	585	13.5	21.34	12.7	21.3
FloatOperation	> 1000	127	425	13.5	6.01	11.7	17.2
FunctionCall	> 1000	89.4	130	12.4	12.48	6.92	5.71
FunctionReturn	> 1000	30.9	98.1	7.66	10.60	5.71	5.37
CFunctionCall	> 1000	N/A	121	8.20	4.42	5.53	9.52
FunctionObjectCall	> 1000	N/A	33.4	10.9	4.42	6.04	
ObjectCreation	2.76	7.43	9.30	0.88	1.61	1.73	0.38
FieldVariable	273	108	183	7.06	4.85	6.88	7.52
MethodCall	145	56.4	68.2		12.6	3.38	3.30

are once transformed to an anonymous function internally, then a compiled form of virtual machine instructions, and finally a evaluated result by executing the transformed anonymous function on a virtual machine.

As described in Section 3.4, a static type checker protects compiled code from unsafe code updates via `eval()`. In addition, we introduce the `Script` class to model the set of global variables and functions. Since the eval function is executed with the association of a Script object, we can isolate unexpected behaviors that result from the execution of eval by switching another script object.

In Konoha, `Script` is a prototye-based singleton class specially designed. The `new Script` generates a new subclass of script and its instance. All functions and global variables are defined as methods and fields respectively in `Script`. If we dispose a `Script` object, all functions and variables over that script are eliminated safely.

```
Script scr = new Script();
scr.eval """
int fibo(int n) {
  if(n < 3) return 1;
  return fibo(n-1) + fibo(n-2);
}
"""
scr.fibo(30);
scr = null;   // dispose the script
```

4.2 Virtual Machine

The code execution engine of Konoha is a register-based virtual machine [11], with a newly designed instruction set. The instructions set ranges from control flows (e.g., call and jmp), to type-specific operations (e.g., iadd and fadd) — similar to CPU instruction sets. Some unique instructions are added: typechk for checking Any value at runtime, box for boxing an integer or a float value, and unbox for unboxing an integer or a float object.

Figure 1 illustrated the complied virtual machine instructions that evaluate the expression `n++`, where the variable `n` is a global variable of `Any` type, requiring dynamic type checking.

Note that integers and floats are usually unboxed because all the arithmetic instructions are designed for unboxed values. However the dynamic type checker needs type information that are recorded at the header of an object. The compiler controls the insertion of boxing/unboxing before the typechk instruction. The static type system helps removing unnecessary box/unbox, or typechk from compiled code,

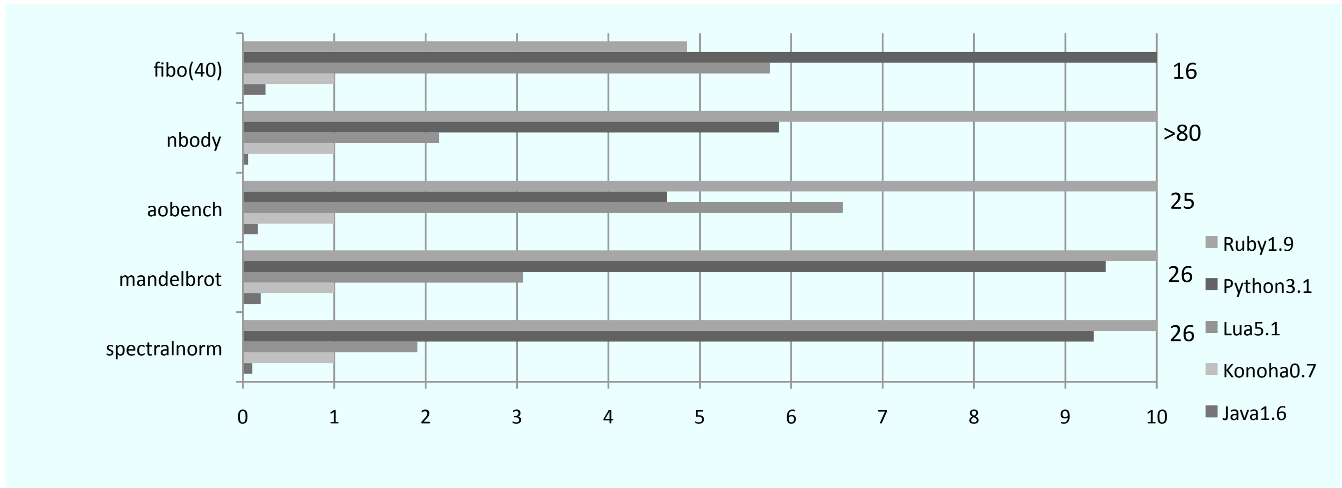

Figure 2. Performance of Java1.6, Lua5.1, Python3.1, Ruby1.9 in relation to Konoha 0.7.

leading to improved performance results, as shown in the next section.

4.3 Performance Study

Good performance is a general concern although it is not our main focus in this paper. In this final subsection, we would like to report our earlier results on the performance study of a statically typed scripting language written in C.

As the development of Konoha is still at the earlier stage, we have used micro benchmark tests mainly designed to improve the virtual machine implementation. Although these tests are primitive and not suitable for the estimation of total application performance, they are well designed to measure each of simple operations such as variable assignment, integer operation, and function calls. Table 2 illustrates the comparison of benchmark scores, obtained by running benchmark tests several times on Mac OS X 10.6 (Core2 2.16Ghz).

In addition to our micro benchmarks, we translated several benchmarks from the Computer Language Benchmark Game [9] into Konoha and compared the result existing benchmarks running on Python 3.1, Lua 5.1 (known as a very fast scripting language), Ruby 1.9.1 and Java 1.6 (depicted in Figure 2). The translated scripts are based on the sources of Java version, not added to any Konoha-specific tuning at the source code level.

Several conclusions can be drawn from this experiment. We found that Konoha enjoyed the performance advantage over major implementations of existing dynamic scripting languages. A runtime type checker is a small part of the whole scripting engine including an object memory allocator and a garbage collector. As shown in the experience of StrongTalk, static types are not necessarily required for achieving good performance. We consider that this result is due to the fact that static typing allows to implement a simpler virtual machine that runs faster.

Compared with Java, Java runs fibo about 3x faster than Konoha, and runs nbody 100x faster than Konoha. We consider that this is mainly because of the lack of program optimization at compilation time. Because the compiler optimization is available on Konoha, we conclude that a static scripting language could provide high-performance execution and ease of programming in the future.

5. Related work

Due to the popularity of JavaScript and Ruby on the Web, scripting languages have received both industrial and academic attention more recently[8]. In this respect, the lack of static types has been regarded as a major pitfall of its language design, resulting in their poor performance and less program robustness. Several attempts have been made to the integration of static typing features into the existing dynamic language[1, 4, 5, 12].

On the other hand, some statically typed languages have increasingly imported dynamic language features to improve the programming experience. Scala has a several features, including type inference and an interactive shell, which are in part available as a scripting language[7]. C# 4.0[13] and Boo[3] introduced `dynamic` and `duck` type, respectively, in order to integrate a runtime type checker into statically compiled code.

We argue that "ease of programming" attributes in scripting languages are not exclusive to dynamic typing. We selected major dynamic language features to model them on top of our newly designed static language. In designing Konoha, we examined a variety of dynamic or static programming languages: Perl, Python, Lua[6], Ruby, Groovy, JavaScript, Thorn[2], Scala[7] and others. The dynamic features of Konoha are influenced primaryly by Python.

6. Conclusion

The lack of static types in scripting languages reveals several difficulties in terms of development such as software evolution and poor performance as its use is largely growing. This paper presented the design and implementation status of Konoha, a static-typing object-oriented scripting language written in C from scratch. Konoha is designed to model dynamic language behaviors, including runtime alteration of object behaviors, duck typing, runtime program updates through the eval function, and execution of incomplete program without compiler stops.

Our implemented Konoha showed good scripting experiences, as well as better performance. Future research direction includes the refinement of static scripting constructors to type theoretic formalization of dynamic object behaviors. Now we are intensively developing Konoha as open source. Any comments and helps are welcome.

Acknowledgments

The work was founded by Japanese Ministry of Economy, Trade and Industry: IPA "unexplored domain of software challenge" program, Japanese Ministry of Internal Affairs and Communications: SCOPE-R for younger researcher fund, a Japan Science and Technology Agency: CREST "Dependable Embedded Operating System for Practical Use" (led by Mario Tokoro). Many people helped the development of Konoha. We also thank the S3-10 reviewers and our shepherd, Carl Friedrich Bolz, who helped us to improve the paper.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *OOPSLA 2007 Proceedings and Companion, DLS'07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64. ACM, 2007.

[2] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the jvm. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 117–136, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: http://doi.acm.org/10.1145/1640089.1640098.

[3] R. B. de Oliveira. The boo programming language. http://boo.codehaus.org/.

[4] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: http://doi.acm.org/10.1145/1640089.1640110.

[5] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, pluggable types for a dynamic language. *Comput. Lang. Syst. Struct.*, 35(1):48–62, 2009. ISSN 1477-8424. doi: http://dx.doi.org/10.1016/j.cl.2008.06.003.

[6] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: http://doi.acm.org/10.1145/1238844.1238846.

[7] M. Odersky. The scala experiment: can we provide better language support for component systems? In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–167, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2.

[8] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.876288.

[9] D. Project. The computer language benchmarks game. http://shootout.alioth.debian.org/.

[10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: http://doi.acm.org/10.1145/1806596.1806598.

[11] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: http://doi.acm.org/10.1145/1064979.1065001.

[12] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 395–406, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328486.

[13] M. Torgersen. New features in c# 4.0. http://code.msdn.microsoft.com/csharpfuture.