Background

- Scripting Languages
  - An interpreter style of program execution
  - From Domain-Specific to General Object-Oriented Programming Features
  - "Ease of programming"
- Popularity and Industrial Acceptance
  - System Operation:
    - Perl, Python, etc.
  - Web Applications:
    - JavaScript, PHP, Perl, Ruby, Python, etc.
  - Game:
    - Lua, Python, etc.

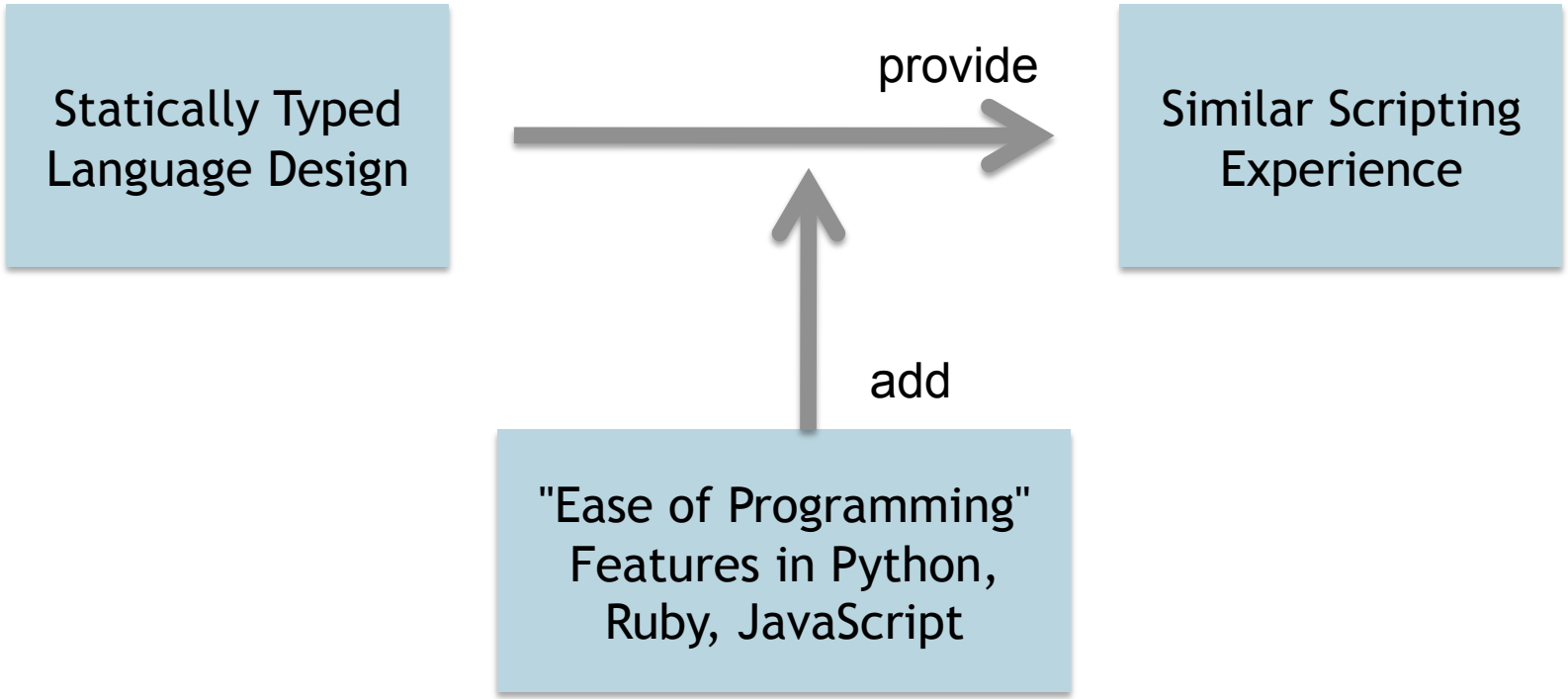
Dynamic Languages

- Major scripting languages adopt dynamic typing
  - enjoy flexibility and easier code reuse
  - enables rapid software update cycle
  - etc..

- Growingly interested in the advantage of static types:
  - earlier detection of programming mistake
  - better documentation
  - good opportunity for compiler optimization
  - program evolution

Our Objective

- **Static Language + Dynamic behaviors = Scripting**



What enables "Ease of Programming"?

- Dynamic Behaviors of Programs
  - very common in scripting languages

Comparisons of Dynamic Behaviors

- **Dynamic Languages share the common programming features on dynamic behaviors**

Features	Perl	Python	Ruby	JavaScript	Lua	Java
typing	dynamic	dynamic	dynamic	dynamic	dynamic	static
field addition	+	+	+	+	+	
method addition			+	+	+	
method rewrite			+	+	+	
missing method			+			
duck typing	+	+	+	+	+	
eval	+	+	+	+	+	
Partial execution	+	+	+	+	+	

+ means end-user's availability, not language support

What enables "Ease of Programming"?

- Dynamic Behaviors of Programs
  - very common in scripting languages

- Probably related language features:
  - Runtime Alteration of Object Behaviors
  - Absence of Type declaration
    - duck typing (easier code reuse)
  - Eval()
  - Execution of Partially Written Programs

Field Addition

- Most scripting languages provide a means to add/modify/ delete the member of an object

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

Person p = new Person("Naruto");
p.age = 17
```

← No declaration of the age field

← The age field is added instead of a type error

- This reduce the cost of rigid class design in programming

Duck Typing

- Type inference provides a complement means to model the absence of type declaration. But,

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

hello(new Person("Naruto"))
hello(new Dog("Hachi"));
```

← Person and Dog is not related

← What type is inferred for the variable w?

Eval() and Partial Execution

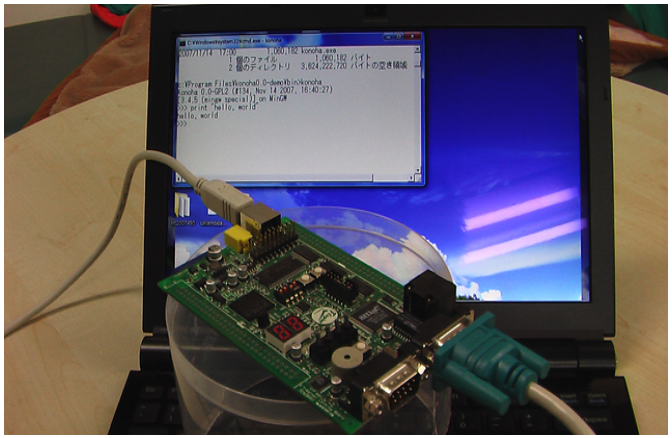
- Eval()
  - taking a string as code, and executing it

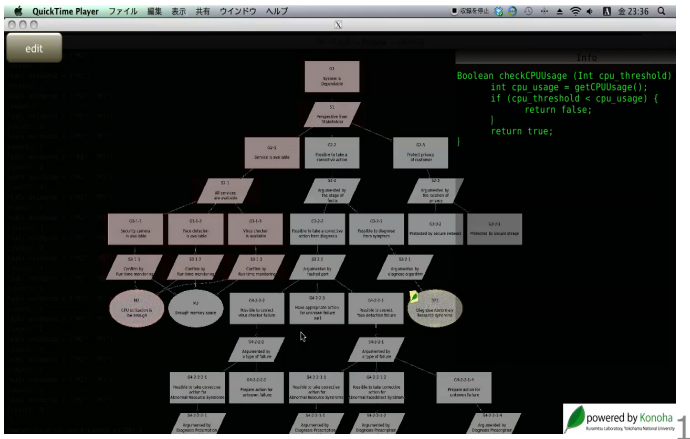
```
n = 10;
n = eval("fibo(n)");
```

- The absence of static type checker:
  - allow the execution of partially correct programs
  - shorten the Edit-Compile-Link-Run process

Konoha – Our Approach

- Konoha 0.7: a new scripting language written in C from scratch
  - Imperative language (C/C++, Java-style grammar)
  - "pure" object-oriented programming, nominal type system, single inheritance, generics
  - Python-style interactive shell
  - open source, running on Linux, Windows, MacOS X, Android OS, and TRON OS.





Language Design

- Java as Lingua Franca
  - At least, Java programmers can read Konoha's source code without misunderstanding
  - Grammar, Classes, Libraries
  - Language support for accepting dialects

    ➔ e.g., toUpper(), ToUpper(), to_upper(),

- Scripting Capability
  - Simplicity ➔ e.g., Int and Float are only supported
  - Rich Operators

    ➔indexing, slicing, inclusion, etc.,

Comparison of Counter Class

(1)
```
class Counter
  def initialize(n)
    @cnt = n
  end
  def count
    @cnt = @cnt + 1
  end
end
c = Counter.new(0)
c.count()
```

(2)
```
var Counter = function(num) {
  this.cnt = num;
  this.count = function() {
     this.cnt++;
  };
}
var c = new Counter(0);
c.count();
```

(3)
```
class Counter:
  def __init__(self, n):
    self.cnt = n
  def count(self):
    self.cnt += 1
c = Counter(0)
c.count()
```

(4)
```
Counter = {}
Counter.new = function(n)
   local obj = {}
   obj.cnt = n
   obj.count = function(self)
      self.cnt = self.cnt + 1;
    end
   return obj
end
c = Counter.new(0)
c:count()
```

(5)
```
package Counter;
sub new {
   my $class = shift;
    my $self = { cnt => 0;};
    return bless $self, $class;
}
sub count {
  my $self = shift;
  $self->{cnt}++;
}
my $obj = new Counter;
$obj->count();
```

(6)
```
class Counter {
  int cnt;
  Counter(int n) { cnt = n; }
  void count() { cnt++; }
}
c = new Counter(0);
c.count()
```

Python-style interactive shell

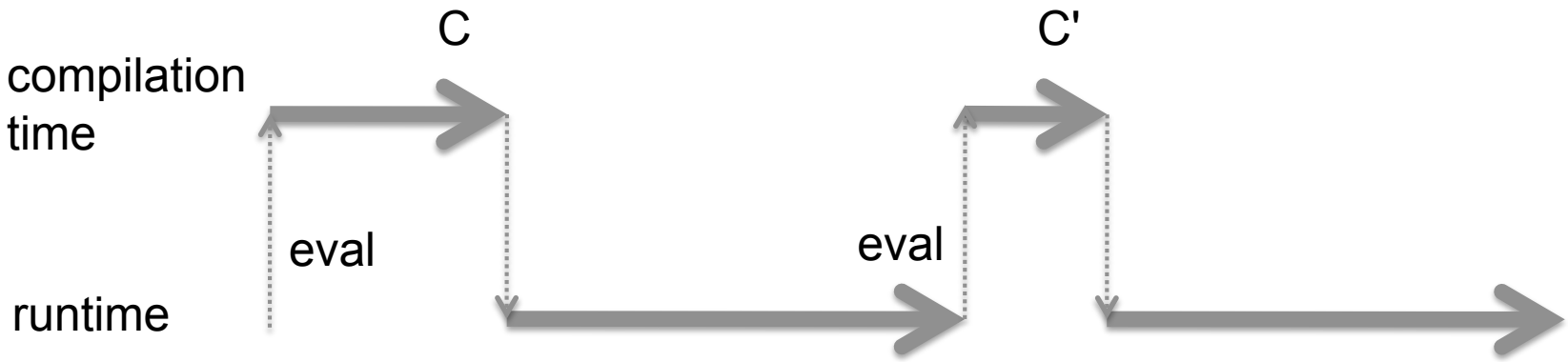
```
$ konoha
Konoha 0.7-beta4(BAKUMAN) LGPL3.0 (rev:1076, Jan  8 2010
12:59:45)
[GCC 4.0.1 (Apple Inc. build 5490)] on macosx_32 (32, UTF-8)
Options: iconv refc sqlite3 thread regex used_memory:235 kb
>>>
>>> class Counter {
..   int count;
..   Counter(int n) { count = n; }
..   void count() { count++; }
.. }
>>> c = new Counter(0);
>>> c.count()
>>> c
Counter {count: 1}
>>>
>>> c.run()
```

Model of Dynamic Behaviors on Static Types

- Runtime Alteration of Objects
- Eval()
  - Must preserve type safety
- Duck typing
  - Dynamic type (➔ an old paper, presented in the paper)
  - Growing type (➔ a new idea)
- Partial execution
  - generate executable code despite detecting type errors

Growing Type

- Key idea
  - Type C is growing at runtime
    - Note that compilation times are parts of runtime
  - Suppose C' is a modified class of C
  - Safety needs C' <: C



Method Addition

- Konoha allows us to define an additional method outside of its class declaration

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

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

← Methods can be added

Method Rewriting and Deletion

- Suppose Person' is a modified class
  - Method rewriting and deletion are defined as the operations satisfying with Person' <: Person

```
class Person {
  String name;
  int age;
  boolean Person.isChild() {
    return this.age < 21;
  }
}
boolean Person.isChild() {
  return this.age < 18;
}
boolean Person.isChild();
```

← The method can be rewritten

← Deletion as an abstract method

Field Addition

- Field accessor is syntax sugar of getter/setter method
  - p.name ➔ p.getName()
  - p.age = 20 ➔ p.setAge(20)
- Explicit field addition can be modeled by the method addition

Field Addition

- Field accessor is syntax sugar of getter/setter method
- Explicit field addition can be modeled by method addition
- Implicit field addition is either
  - type error, expanded fields in all objects, or added metadata entry

```
class Person {  String name; }
Person p;

p.age = 17;              // type error

dynamic p.age = 17;      // added age to all instances

virtual p.age = 17;      // added metadata entry
                         //  p.meta["age"] = 17
```

Dynamic Any Type

- Dynamic any type is a special type that allows an additional runtime type check instead of a static type check

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

← Not statically checked.
Instead, insert a runtime type check before calling w.name?

Growable Type

- Growing type allows runtime type growing
  - Suppose NameLike is an undeclared but growable class

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

← If .name is not found in NameLike, the compiler adds getName()

- This could model the "duck typing" features in a both static and nominal type sense

"Run Anytime" Compilation

- Correct parts of program must be executed despite the detection of type errors

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

← Detected type error

"Run Anytime" Compilation

- Correct parts of program must be executed despite the detection of type errors

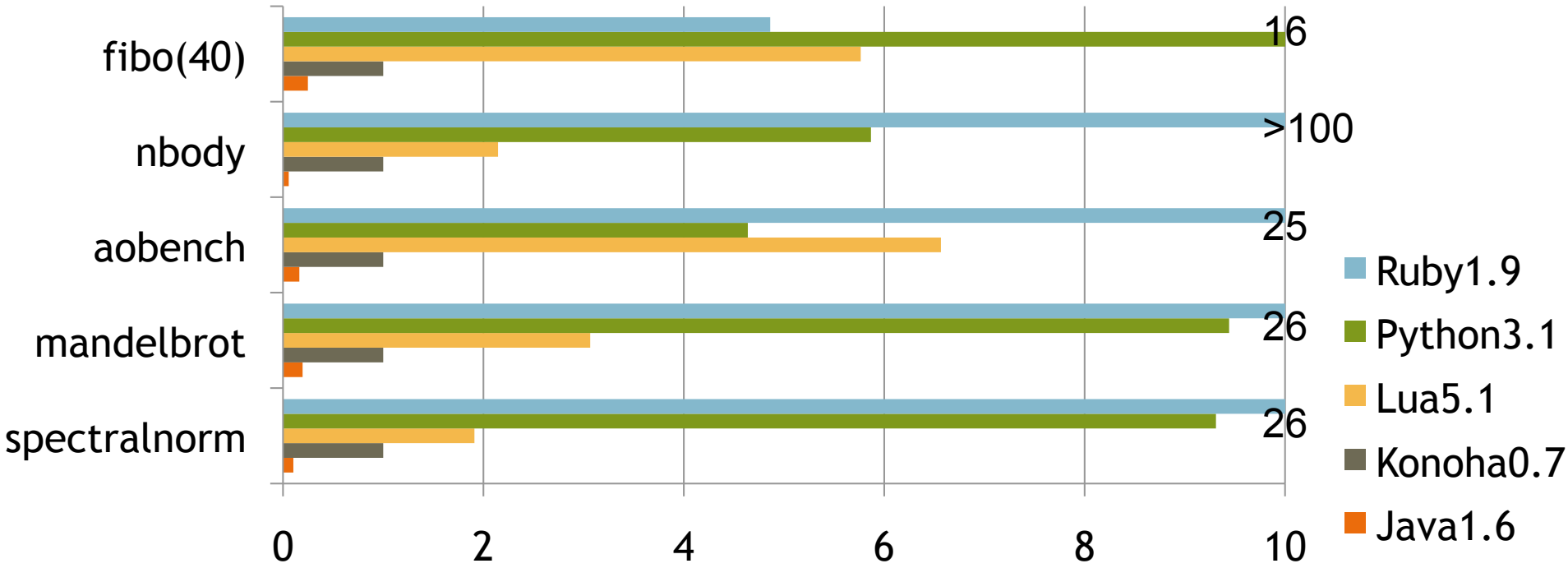
```
int newSerialNum (int n)
{
  if(n == 0) {
    throw new TypeError("");

 }
 return n+1;
}
```

←Replace that block with code throwing a runtime exception

- Now we can correctly run the function except for newSerialNum(0)

(Preliminary) Performance Study



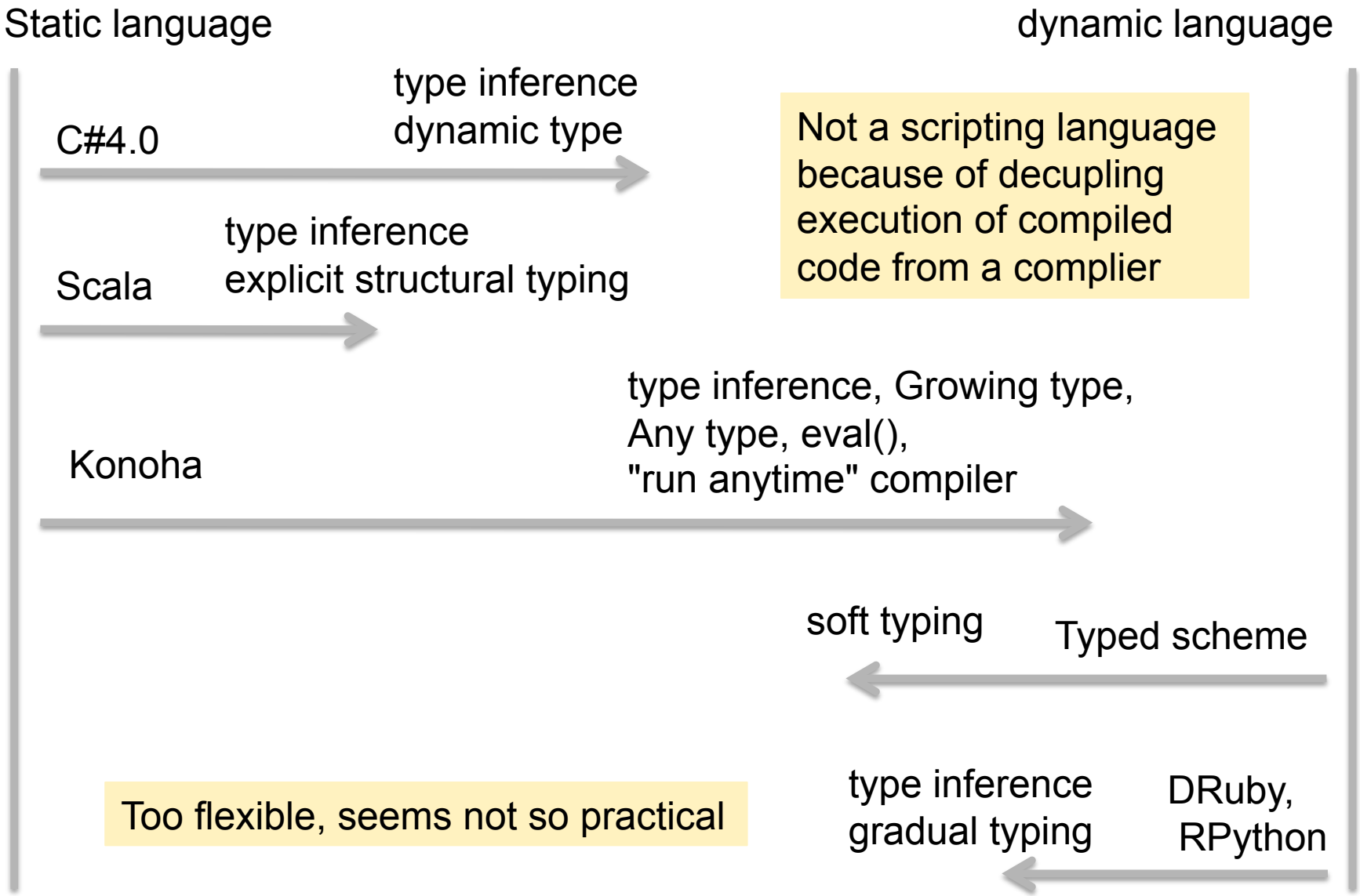
- **Intuitive insights:**
  - **Java1.6 : Konoha0.7 = 1: 8**
  - **Konoha0.7 : Python3.0 = 1 : 8**
- **Future: Konoha-JIT x2 Konoha-SSA x?**

Comparisons of Dynamic Behaviors

	Python	Ruby	JavaScript	Java	Scala	Konona
type declaration	dynamic	dynamic	dynamic	static	static	static
field addition	+	+	+			+
method addition		+	+			+
method rewrite		+	+			+
missing method		+				abstract
duck typing	+	+	+		structural typing	dynamic
eval()	+	+	+			+
Partial execution	+	+	+			+

- **Scala is used as a static scripting language, but it is still far from dynamic languages in terms of dynamic behaviors**

Related Work



Conclusion

- Konoha is a statically typed scripting language that provides the same or very similar programming experiences with existing dynamic languages, such as Python and Ruby.

- Konoha enjoyed static features:
  - Readability for Java programmers
  - Our earlier implementation shows good performance
- Our approach is practical

- Future Work
  - More formal discussion on type theory
  - Improve open source products

Thank you for your attention

- Konoha is available at the following site:
  - http://konoha.sourceforge.jp/
  - http://code.google.com/p/konoha