Typing Ruby Programs

Soutaro Matsumoto
(University of Tsukuba)

牛久大仏

RubyKaigi'08

soutaro

- Graduate student in University of Tsukuba
  http://d.hatena.ne.jp/soutaro
- LLRing (Aug, 2006) - Language Update (OCaml)
- RejectKaigi 2007 (June, 2007) - Pragger is LL
- IPSJ PRO 2007-3 (Oct, 2007) -
  Type Inference of Ruby Programs Based on Polymorphic Record Type (多相レコード型に基づくRubyプログラムの型推論)

Outline

- Ruby Programming Language
- Typing Ruby Programs
- Intuitive Overview of the Type Inference Algorithm
- Limitations
- Conclusion

Ruby Programming Language

```
class A
  def f(x, y)
    x.+(y)
  end
end

a = A.new()
a.f(1, “2”)
```

Ruby is ...

- an untyped language
- an Object Oriented Language
- an dynamic language

Ruby is an Untyped Language

```
class A
  def f(x, y)
    x.+(y)
  end
end

a = A.new()
a.f(1, “2”)
```

- It is clear that the program results TypeError
- Ruby implementations do not reject such programs
- There are no type declarations in Ruby programs
- Most programs do not want NoMethodError, TypeError, LocalJumpError, ArgumentError, or .... ?

Ruby is an OO Language

```
class A
  def f(x, y)
    x.+(y)
  end
end

a = A.new()
a.f(1, “2”)
```

- Ruby has object / self / class / fields / inheritance
  (fields, self, and inheritance are important from the point of view of type system)
- Most programs are written to support structural conformance (so-called duck typing)
- Some programs are not: Object#is_a?

Ruby is a Dynamic Language

```
class A
  def f(x, y)
    x.+(y)
  end
end

a = A.new()
a.f(1,
    2.__send__(:to_s))
```

- eval, __send__, define_method, lambda, method_missing, block_given?, ..., and more seem harmful
- Extension libraries are nightmarish
  They are not written in Ruby but C
  It is impossible to guess what happens when programs call methods provided by extension libraries

Typing Ruby Programs

- To statically reject programs that may result runtime errors such as NoMethodError, ArgumentError, or ...
- No extra type annotations
  Existing Ruby programs should type check
- Dynamic features can not be ignored
  Don't just reject them

Type Inference

- Reconstruct types of expressions from their contexts
- Based on ML style type inference
  By unification and polymorphic record
  (like OCaml / SML#)
  - cf. Flow analysis
    So called Safety Analysis

Dynamic Features

- Extension of existing classes
- Reflections / eval

Extension of Existing Classes

```
class String
  def parenthesis()
    “(#{self})”
  end
end

“a”.parenthesis()
```

- Override / add methods in existing classes (open class)
- Most Ruby programmers love it Widely used in Ruby programs
- One of the most characteristic feature of Ruby

Reflections / `eval`

- It is impossible to type them
- Introduces type annotation [not yet]
  Type annotations should be written in Ruby's syntax
- `dynamic_cast(Integer, eval("1"))` [not yet]
  C++ like syntax
- How to treat parametrized types (like Array)

Intuitive Overview of Type Inference Algorithm

- Definition of Type
- Examples

Definition of Type

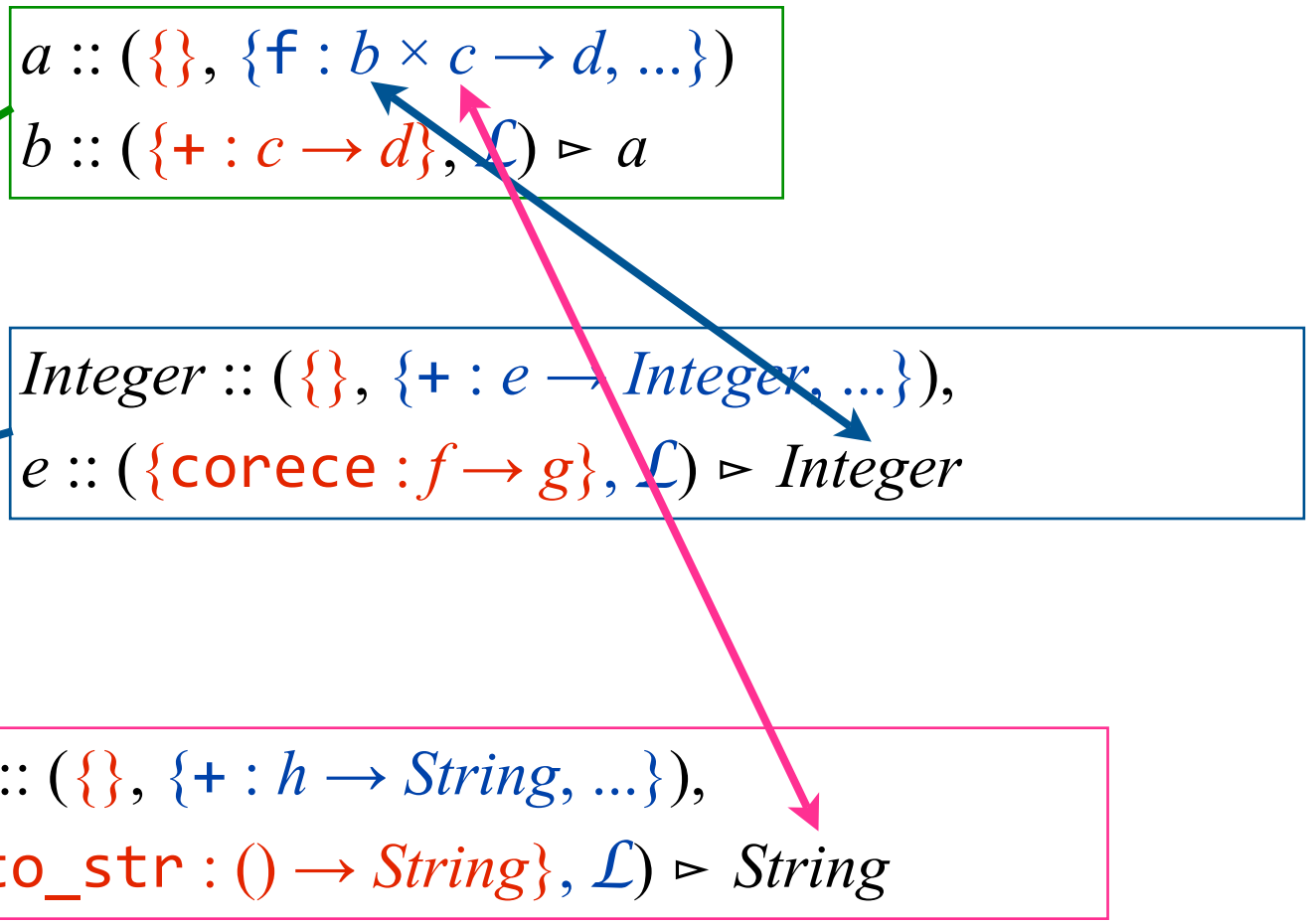
- Type is a pair of set of methods
- Required methods / Available methods
- All types should hold the following:
  Required methods $\subseteq$ Available methods

$$a :: (\{\texttt{puts} : () \to \texttt{String}\}, \{\texttt{puts} : () \to b, \ldots\}) \rhd a$$

Reads:
The type is a, which should have method puts
and have methods puts, ...

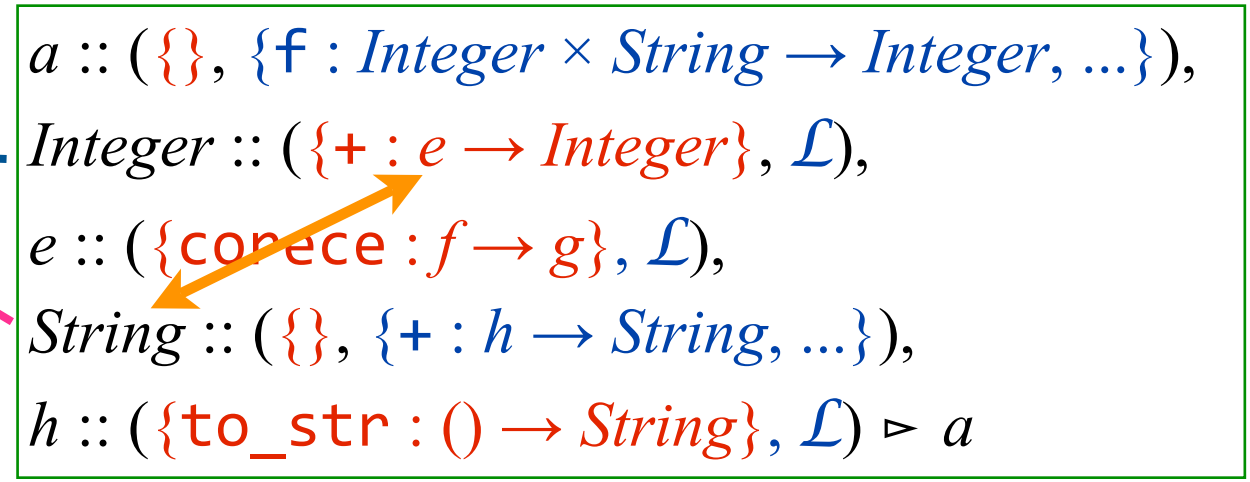
Example

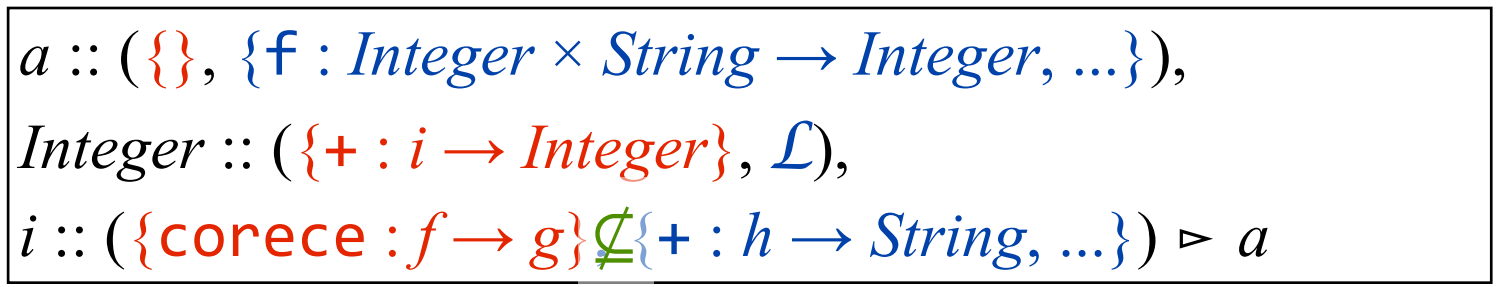
```
class A
  def f(x, y)
    x.+(y)
  end
end

a = A.new()
a.f(1, "2")
```



Example

```
class A
  def f(x, y)
    x.+(y)
  end
end

a = A.new()
a.f(1, “2”)
```






```
-:8: undefined method {`coerce'} (NoMethodError)
```

Limitations

- Polymorphic methods
- Heterogeneous collections
- Soundness
- `Array#map`
- `lambda`

Polymorphic Methods

```
class A
  def id(x)
    x
  end
end

def huga(a)
  a.id(1) + 1; a.id(“a”) + ”b”
end

a = A.new
a.id(1) + 1; a.id(“a”)

huga(a)
```

- Methods can not have polymorphic type
- Method parameters can not have polymorphic type

Heterogeneous Collections

```
a = [1, “2”, []]

a.each {|x|
  puts x.size
}

a[0] + a[1].to_i
```

- Iterations are supported (in most cases)
- Elements only have methods common for all elements (`size`)
- Arrays do not have `to_i` method

  ⇒

  `a[0]` nor `a[1]` can not be typed to have `to_i` method available

Soundness

```
x = 100
10.times {
  if x % 2 == 0
    x = "101"
  else
    x = 101
  end
}
```

- Sound type inference algorithm ensure that typed programs never go wrong
  ONE OF THE MOST IMPORTANT PROPERTIES !!!
- Our algorithm does not have that property: it is not sound
- Soundness is not so important
  Ruby is a safe language itself
  To support many programs is more important

Array#map

```
[1,2,3].map {|x|
  x.to_s
}.map {|x|
  x.to_s
}.map {|x|
  x.size
}
```

The result of map is typed as the same

lambda

```
def hoge()
  lambda
end

p = hoge {|x| puts x}
p.call(1)
```

- It looks like an ordinal method, but in fact it is a primitive (Ruby programs can not simulate `lambda`)
- `lambda` without block captures block given to outer method
- Need a new rule to support it
- But with a warning !!! (Good news for me)

ToDo

- Improve type inference
  Many features are unsupported
- Formalize the semantics of Ruby
  The semantics of the most unsupported features are unclear (for me)

Conclusion

- http://www.typing-ruby.org
- Type checking is your friend
  And it is very interesting
- `eval`, `send`, `define_method`, `lambda`,
  Rails, and more are my enemy