StrongRuby : gradual typing for Duby.

Pedro Del Gallego

Abstract :

Static and dynamic type systems have well-known strengths and weaknesses. Statically typed programming languages incorporate a built-in static type system. Every legal program must successfully typecheck according to the rules of the type system.

Gradual typing provides the benefits of both static and dynamic checking in a single language by allowing the programmer to control whether a portion of the program is type checked at compile-time or run-time by adding or removing type annotations on variables. , that have no effect on the dynamic semantics of the language.

Introduction :

Static and dynamic typing have complementary strengths, type safety, performance. In the oder hand, dynamic typing brings adapatability.

Many dynamic languages allow explicit type annotations. Common LISP is an example. In Common LISP, adding type annotations improves performance but the language does not make the guarantee that annotating all parameters in the program prevents all cast errors at run-time.

As is the case for gradual typing I mention few more related works here. Anderson and Drossopoulou developed a gradual type system for BabyJ [2] that uses nominal types. The new Javascript 2 specification [3] also describe gradual types for type checking. The Strongtalk system [4] is a type checker for Smalltalk code that relies on programmer annotations for types, but erases all types and runs the underlying Smalltalk code at execution time .

The goals of Gradual type are :

- Programmers may incrementally add type annotations to gradually increase static checking.
- Gradual typing has no effect on the semantic of the programming language.
- Does not mandate type annotations in the syntax.

Starting with an untyped program built on ad hoc extensible objects and hand-coded data validation, a typical Ruby program, we can apply type annotations at key points to make validation faster and more reliable.

Variables can optionally be annotated with type constraints, and an implementation can perform static type checking and early binding.

If the program is written ian *strict mode,* where every possible variable or method is typed, all the type error should be catched in advance.

Notes on the programmer's side :

Code is worth than a thousend words.

Model 1 : Full Dynamic

```
class PlanarPoint
  attr_accessor :x, :y

  def move_x(offset)
    self.x+= offset
  end
end
```

Model 2: Partial static

```
class PlanarPoint
  attr_accessor :x, :y, Int

  def move_x(offset): ? -> Self
    self.x+= offset
  end
end
```

Model 3 : Strict mode

```
class PlanarPoint
  attr_accessor :x, :y, Int

  def move_X(offset): Int -> Self
    self.x+= offset
  end
end
```

The programmer start writting in a full dynamic mode. When he feels that is needed, then he can start incrementally using annotations types in those portion of the program that he need to be type safe or a better performance. At the end he can use a strict mode to made his code more realible or faster, or he can go backward and remove all them. Just as untyped code can become gradually typed as need dictates,

The above code demostrate the behavior of each case.

Full Dynamic

```
a: int = 1
point = Point.new

# ok
p.move_x(a)
p.move(1)

# Dynamics Type Errors
p.move_x(“b”)
p.move_x(a)+”hola”
```

Partial static

```
a: int = 1
point = Point.new

# ok
p.move_x(a)
p.move(1)

# Dynamics Type Errors
p.move_x(“b”)

# Static Type Errors
p.move_x(a)+”hola”
```

Strict mode

```
a: int = 1
point = Point.new

# ok
p.move_x(a)
p.move(1)

# Static Type Errors
p.move_x(a)+”hola”
p.move_x(“b”)
```

Notes on the implementator's side :

We say that two types match if they both are known and are the same type.

Parameters with no type annotation are given the dynamic type ‘?’ . The type ? is represented in the language by the top parent, in ruby that means the Object type.

Types :

- Every class define a new type : Int, Boolean , String, Float, and every programmer class
- Any type allowed : ? , what really means Object
- Union types : (Int, String), (TrueClass, FalseClass), (Cat, Dog)
- Generics :
  - Arrays :[Int], [Person],
  - Hash : {Int, Person}
- Special types : Self, means the same type of the current object.

Predefine union types : The language predefines several convenient union types:

- type Number = (Fixnum, Numerical, BigDecimal, Float)
- type FloatNumber(Double, Float)

- type Boolean = (TrueClas,FalseClass)

Type annotations and type checking: Type annotations on locations (whether variables, parameters, or properties, and irrespective of the binding form or the binding object) act as run-time assertions, type checks, on the types of values stored into those locations, whether by the program or by the implementation. If a type check fails, the implementation throws a TypeError exception.

Function and methods types : Types describe functions and methods. The declaration :

```
def foo (a , b): int, string -> boolean
   #.....
end
```

Describes a Function object that takes two arguments, one int and the other string, and which returns a boolean. (This is only a proposal, we must experiment and ask for feedback for other kind of notation). i.e other possible notation.

```
def foo (a:int  , b: string ): boolean
   #.....
end
```

Or use the actual duby notation

```
def foo (a, b): boolean
   {:a=>:fixnum, :b=>:fixnum}
   #.....
end
```

Subtypes : An object that has type T also has type U if T is a subtype of U, T is a subtype of U if T is more specific than U, that is, T describes fewer values than U because T has more constraints. A key point of subtyping is that if a variable is annotated with a type U then it will also accept any value of type T if T is a subtype of U.

Without getting technical or exhaustive, the following rules for subtyping hold:

- Any type is a subtype of itself
- Any type is a subtype of ?
- The only member of the type null is the value null, and the only member of the type undefined is the value undefined
- If D is a subclass of B then D is a subtype of B
- If C include the module I then C is a subtype of I

Why Duby instead of Rubinius, JRuby or MRI:

Duby have implemented a compiler written in Ruby, that would and made easy to customize it. Also, Duby and StrongRuby will require their richer AST, both need to support type information, method signatures among other metadata information. Them any impovement in the Duby AST will improve the also StrongRuby AST. and viceversa.

Schedule :

Steps :

1. Familiarize myself with the Charles Nutter's bytecode generating DSL
2. Familierize myself with the Ruby compiler from Duby.
3. Parse and Compiler improvement.
4. Extend the AST tree to support more information.
5. Type checker system.
    1. Write test set for type checker system.

2. Type Checker system infrastructure. (tags, ...)
3. Add check between basic types.
4. Add check subtypes.
5. Method chain inference types.
6. Add Array and

6. Early bindings (Out of the scope of Google Summer of code Program)
   1. Ruby built-in “primitives” classes to Java primitives.
   2. Binding to user classes.

Benefits to the community :

I think, push ideas to ruby-like language from academical papers and other languages is a healthy exercise for the ruby community. This proposal promote the debate on adding some kind of static types to Ruby.

At the same time this project can be a good proof of concept of how good (or bad) gradual typing plays good Ruby.

Alongside to writing all the necesaries mechanism. I will work in the ruby compiler written by Charles Nutter. Probably I will need to find bugs and i will try to fix them. Imporving this compiler probably make easy to develop new proof of concept with ruby.

More ideas on this topic.

Compiler modes : The compiler can work in three modes:

1. The standart Ruby mode : The compiler ignore all the annotations
2. The strongtalk mode : Use annotations informaton in the compile time to perform type check, but remove this information.
3. The full mode. Use annotations informaton in the compile time to perform type check, and use the information increase the performance

Type inference: Using the annotations information , it should be possible to implement a complete Type Inference system.

Bibliography :

[1] **Gradual Typing for Objects,** [PDF]
http://www.ecmascript.org/es4/spec/overview.pdf

[2] **BabyJ: From Object Based to Class Based Programming via Types,**
http://citeseer.ist.psu.edu/570376.html

[3] **Proposed ECMAScript 4th Edition – Language Overview**, [PDF]
http://www.ecmascript.org/es4/spec/overview.pdf

[4] **StrongTalk : Typecheking smalltalk for production environment.**