Diamondback Ruby Guide

Mike Furr, David An, Jeff Foster, Mike Hicks

April 16, 2009

Contents

1 Introduction

Ruby is a dynamically typed, object oriented scripting language. Dynamic typing keeps the language flexible, allowing small, expressive programs to be written with little effort. However, software never stands still. As these programs evolve, the lack of static typing makes it increasingly difficult to maintain, extend, and reason about. Diamondback (DRuby) is a tool that augments Ruby with a static type system. DRuby's type system was designed to handle common idioms used by Ruby programmers so that programs would not need to be changed in order to gain the benefits of static checks.

2 Installation

2.1 Requirements

To build DRuby from source, you will need the install the following dependencies.

2.1.1 OCaml Dependencies

These programs are should be available with most linux distributions. For other operating systems, the package manager *godi* (http://godi.camlcity.org/godi) can provide a convient way of installing them if you don't wish to do so manually.

- ocaml(3.09.2 or higher)
- ocamlfind
- ounit
- omake
- ocamlgetopt
- ocamlgraph

2.1.2 C Dependecies

This library is also commonly distributed with linux distributions. It is also available through the MacPorts package manager for OS X.

- syck

2.2 Building and Installing

DRuby uses the OMake build system to build it from source. OMake is a make-like build system that includes both configuration and build rules. Thus, the build is broken into three steps: configuration, compilation, and installation. However, these steps can also be combined into a single command:

```
$ [sudo] omake –config VAR1=arg1 VAR2=arg2 install
```

Here, we call omake with the –config flag to set the variables VAR1 and VAR2 and tell it to build the install target. The install target depends on the configuration and compilation targets and will therefore do all the necessary work. The call to sudo is needed if you do not have write permissions to your Ruby installation. Alternatively, druby can be installed outside of the system directories using the variables below. The current list of supported build variables are:

- PREFIX - The path used to provide defaults for other DRuby paths such as where to install the druby binary. Defaults to /usr/local/
- BINDIR - The install location of the druby binary. Defaults to PREFIX/bin
- SYSCONFDIR - The install location of the druby global configuration file, druby.conf. Defaults to PREFIX/etc
- DESTDIR - A staging directory for building packages of DRuby. Any necessary paths will computed based on PREFIX, but the actual installation step will stage the installation into DESTDIR/PREFIX/....
- RUBYLIB - The location of your Ruby installation (library files). For example, /usr/lib/ruby/. You should not have to specify this under normal circumstances as it will be determined by the 'ruby' executable found in your PATH.
- RUBYSITEDIR - The location of your Ruby installation's site-lib directory. For example /usr/lib/ruby/site-lib. Like RUBYLIB, this should be automatically computed for you.
- DRUBYSITELIB - The install location for DRuby's runtime ruby files. For example, /usr/lib/ruby/site-lib/1.8/druby/. Automatically computed based on RUBYSITEDIR
- DRUBYLIB - The install location for DRuby's non-ruby files. For example, /usr/lib/ruby/druby/. Automatically computed based on RUBYLIB
- SYCK - The base location of the syck C library. For example, this should be specified as /opt/local if libsyck* is in /opt/local/lib/

Note: if you need to change these variables after attempting to build DRuby, you must specify the command line argument: –config. As an example, a user on OS X with MacPorts and godi installed would build and install DRuby with:

```
$ sudo omake --config PREFIX=/opt/local SYCK=/opt/local install
*** omake: reading OMakefiles
--- Checking for ocamlfind... (found /opt/godi/bin/ocamlfind)
--- Checking for ocamlc.opt... (found /opt/godi/bin/ocamlc.opt)
--- Checking for ocamlopt.opt... (found /opt/godi/bin/ocamlopt.opt)
--- Checking whether ocamlc understands the "z" warnings... (yes)
--- Checking ocaml-getopt... (/opt/godi/lib/ocaml/pkg-lib/getopt)
--- Checking ocamlgraph... (/opt/godi/lib/ocaml/pkg-lib/ocamlgraph)
--- Checking for gcc... (found /usr/bin/gcc)
--- Checking for g++... (found /usr/bin/g++)
--- Checking for syck.h... (found)
--- Checking for ruby... (found /opt/local/bin/ruby)
--- Checking Ruby libdir... (/opt/local/lib)
--- Checking Ruby ruby_version... (1.8)
--- Checking Ruby sitelibdir... (/opt/local/lib/ruby/site_ruby/1.8)

Installation summary:
              binary: /opt/local/bin/druby
         config file: /opt/local/etc/druby.conf
          ruby files: /opt/local/lib/ruby/site_ruby/1.8/druby
       support files: /opt/local/lib/ruby/druby/1.8
...
```

DRuby has a couple of other useful targets:

- config - this target simply prints out the current configuration variables
- .DEFAULT - build the entire DRuby project and run the test suite. This is also the what is built if no target is specified.

3 Running DRuby

In order to make using DRuby as easy as possible, the druby binary can be used as a drop in replacement for the Ruby interpreter. However, instead of interpreting the ruby code, it performs its static analysis. DRuby accepts a superset of the command line arguments that Ruby accepts. For instance, one can invoke:

```
$ druby -I my_dir -rmylib filename.rb
```

And DRuby will act appropriately (adding *my_ dir* to its seach path, and preloading the *mylib.rb* file). It is also possible to have DRuby invoke the regular Ruby interpreter after it has finished its analysis by using `--dr-run-ruby` option.

```
$ druby --dr-run-ruby filename -- args
```

All of DRuby's command line arguments are prefixed by –dr-. To ensure any program arguments do not conflict with these arguments, it is recommended that you pass any arguments intended for your script after the `--` delimiter.

DRuby's command line arguments are FIXME

These arguments can also be specified via configuration files. DRuby looks for options in the following order. A latter declaration overrides a previous one:

- The global configuration file: SYSCONFDIR/druby.conf
- A per-user configuration file: $HOME/.druby.conf
- A file called druby.conf in the current working direcotry
- Arguments passed on the command line to druby

4 A small example

```
args = ARGV
sum = 0
args.each do |v|
  sum += v
end
puts sum
```

Consider the example shown above which prints the sum of the command-line arguments. Try DRuby on this program to see whether this code is well-typed or not. You will get a similar error message as the following:

```
$ druby --dr-run-ruby first.rb
[ERROR] instance String used where Numeric expected
          It does not support methods ~, |, zero?, truncate, to_int, times, step,
          singleton_method_added, rpower, round, remainder, rdiv, quo, prec_i, prec_f,
          prec, power!, numerator, nonzero?, nan?, modulo, lcm, integer, infinite?,
          id2name, gcdlcm, gcd, floor, finite?, downto, divmod, div, denominator, coerce,
          chr, ceil, abs, ^, >>, /, -@, -, +@, **, &
  in typing method call sum.+ at ./first.rb:4 in typing method call sum.+ at ./first.rb:4
  in typing method call args.each at ./first.rb:3
  in assignment to ::ARGV at /opt/local/lib/ruby/druby/1.8/base_types.rb:2993
  in creating instance of String at /opt/local/lib/ruby/druby/1.8/base_types.rb:2993
  in typing expression %{args} at /opt/local/lib/ruby/druby/1.8/base_types.rb:2993
  in typing actual argument %{args} at /opt/local/lib/ruby/druby/1.8/base_types.rb:2993
  in typing ::Array.new at /opt/local/lib/ruby/druby/1.8/base_types.rb:2993
DRuby analysis complete.
now running Ruby...
ruby  first.rb  0
```

The short description (the top most sentence) of the error message says, "`instance String used where Numeric expected`," and the context information (filenames and line numbers) first points to `sum.+` method at line 4 of `first.rb`. By inspecting the code, we realize that the command line arguments stored in ARGV are strings not numbers and therefore trying to add a string to a fixnum (sum) is the cuase of the error. Change `sum += v` to `sum += v.to_i`, and run DRuby again. You will see that the program is now accepted. This program is now well-typed because `to_i` method converts the receiver to an integer.

The long description of the message (the message beginning from "`It does not...`" to right before the first context information), tells us more details about the error. Since Ruby uses structural typing (duck typing),

DRuby models types in a samilar manner. Despite having different class names (String vs. Numeric), DRuby tries to treat a String as a Numeric. It is safe to use a String in place of a Numeric if and only if String responds to at least as many methods as Numeric. However, this is not the case, and so DRuby reports each missing method.

Following this line is a list operations that DRuby performed or observed that it believes contributed to the error. As DRuby walks over a program, it generates structural constraints on objects and methods in order to discover any potential errors. These constraints form a graph, and an error represents *some* inconsistent path through this graph. Unfortunately, it can be fairly difficult to construct a reasonable error message from this graph in general, and DRuby currently uses a heuristic to attempt to produce a useful error message. However, this process is not perfect. Sometimes DRuby will print the same information twice, or print a series of constraints that are seemingly unrelated. If you encounter such a situation, please feel free to email us with the code that caused the error so we can try to improve the heuristic.

You may be wondering how DRuby is capable of typing `ARGV, Fixnum#+, or String#to_i` in the first place. Because Ruby core library is mostly written in C, we cannot directly analyze it. Instead, DRuby uses a stub file called, `base_types.rb`, which defines stub classes and modules along with appropriate type annotations. These annotations are essential to our analysis because they define the types of the classes and modules that are built into the interpreter. `base_types.rb` can be found in `druby/`*`RUBY_VERSION`* directory inside Ruby's library directory (usually `/usr/lib/ruby`). For more details on the type annotation language, refer to Section 5.

5 Type Annotation Language

Although Ruby gives the illusion that built-in values such as 42 and true are objects, in fact they are implemented inside the Ruby interpreter in C code, and thus DRuby cannot infer the types of these classes. However, we can declare their types using DRuby's type annotation language. In DRuby, type annotations appear before the corresponding class or method declaration. All annotations appear on a line beginning with `##%`, and therefore appear as comments to the standard Ruby interpreter. These annotations are currently parsed with the "def" or "class" tokens, so nothing (except white-space) should appear between an annotation and these keywords.

Annotations can also be used to describe Ruby code, not just C code. If an annotation is placed on a non-empty method, the body of the method will be checked to ensure it satisfies the annotation (assuming –dr-check-annotations is set). The one exception is for intersection types, which are not yet checked statically (it's a work in progress).

Here is part of the declaration of class String:

```
class String
  ...
  ##% "+" : (String) -> String
  def +(p0); end
  ##% insert : (Fixnum, String) -> String
  def insert(p0, p1); end
  ...
end
```

The first declaration types the method `+` (non-alphanumeric method names appear in quotes), which concatenates a `String` argument with the receiver and returns a new `String`. Following the annotation is a dummy method body that is empty and therefore ignored by DRuby. Similarly, the next annotation line declares that `insert` takes a `Fixnum` (the index to insert at) and another `String`, and produces a new `String` as a result.

5.1 Intersection Types

Many methods in the standard library have different behaviors depending on the number and types of their arguments. For example, here is the type of `String`'s `include?` method, which either takes a `Fixnum` representing a character and returns true if the object contains that character, or takes a `String` and performs a substring test:

```
...
##% include? : ( Fixnum) -> Boolean
##% include? : ( String ) -> Boolean
def include?(p0); end
...
```

The type of `include?` is an example of an intersection type. A general intersection type has the form t and t', and a value of such a type has both type t *and* type t'. For example, if `A` and `B` are classes, an object of type `A` *and* `B` must be a common subtype of both `A` and `B`. In our annotation syntax for methods, the *and* keyword is omitted (only method types may appear in an intersection), and each conjunct of the intersection is listed on its own line.

Another example of intersection types is `String`'s `slice` method, which returns either a character or a substring:

```
##% slice : ( Fixnum) -> Fixnum
##% slice : ( Range) -> String
##% slice : ( Regexp) -> String
##% slice : ( String ) -> String
##% slice : ( Fixnum, Fixnum) -> String
##% slice : ( Regexp, Fixnum) -> String
def slice(p0, p1=nil); end
```

Notice that this type has quite a few cases, and in fact, the Ruby standard library documentation for this function has essentially the same type list. Intersection types serve a purpose similar to method overloading in Java, although they are resolved at run time via type introspection rather than at compile time via type checking. Note that our inference system is currently unable to infer intersection types, as method bodies may perform ad-hoc type tests to differentiate various cases, and so they can currently be created only via annotations.

In order to determine what part of an intersection type to use for a given method call, DRuby tries to match the types of the arguments of the caller to list of possibilities. Because DRuby performs inference, it may not now which classes are subtypes of each other at this point. Therefore, it uses a very simple check: name equality. Consider the following example:

```
class A; end
class B < A; end
##% f : () -> Boolean
##% f : A -> Boolean
def f(x=A.new) true end
f(A.new) # type checks
f(B.new) # fails
```

Unfortunately, the latter call to f() fails because DRuby can not yet tell if B is a subtype of A (perhmaps more method defs are coming below!) This can be fixed using an explicit subtype annotation, as discussed in Section 5.9

5.2 Optional Arguments and Varargs

One particularly common use of intersection types is methods with optional arguments. For example, `String`'s `chomp` method has the following type:

```
##% chomp : () -> String
##% chomp : (String) -> String
def chomp(p0=$/); end
```

Calling `chomp` with an argument s removes s from the end of self , and calling chomp with no arguments removes the value of (global variable) `$/` from `self` . Since optional arguments are so common, DRuby allows them to be concisely specified by prefixing an argu- ment type with `?`. The following type for `chomp` is equivalent to the intersection type above: `chomp : (?String) -> String`

DRuby also supports varargs parameters, specified as `*t`, meaning zero or more parameters of type `t` (the corresponding formal argument would contain an `Array` of `t`'s). For example, here is the type of `delete`, which removes any characters in the intersection of its (one or more) `String` arguments from `self` :

```
##% delete : (String, *String) -> String
def delete(p0,*p1); end
```

Notice this type is equivalent to an intersection of an unbounded number of types.

5.3 Union Types

Because Ruby is dynamically typed, it is easy to freely mix different classes that share common methods. For example, consider the following code:

```
class A def f() end end
class B def f() end end
x = if ... then A.new else B.new end
x.f
```

Even though we cannot statically decide whether or not `x` is an `A` or a `B`, this program is clearly well-typed at run time, since both classes have an f method. To support this kind of coding, DRuby supports union types of the form *t* `or` *t'*, where *t* and *t'* are types (which can themselves be unions). For example, `x` above would have type `A` or `B`, and we can invoke any method on `x` that is common to `A` and `B`. The example below shows how a union type is used in `base_types.rb`. Here `%` method takes either `Fixnum`, `Float`, or `String` and gives back a `String`.

```
...
##% "%" : (Fixnum or Float or String) -> String
def %(p0); end
...
```

5.4 Object Types

While most types refer to particular class names, Ruby code usually only requires that objects have certain methods, and is agnostic with respect to actual class names. The code snippet below shows the `print` method, which takes zero or more objects and displays them on standard output. The `print` method only requires that its arguments have a no-argument `to_s` method that produces a `String`. In this case, the

print method can be annotaed so that it only requires that the argument has `to_s` method instead of the argument being a `String`. The object type must be in the following form, $[m_0 : t_0, ..., m_n : t_n]$, where each m_i has type t_i.

```
module Kernel
  ...
  ##% print : (*[to_s : () -> String]) -> NilClass
  def print(*p0); end
  ...
end
```

5.5 Self Types

In some cases, it is cumbersome to give a nominal or structural type to an object, especially to a return object. Consider the following code snippet:

```
class A; def me() self end end
class B < A; end
b = B.new.me
```

Here class `A` defines a method `me` that returns `self`. If we give `me` the type () -> A, DRuby does not treat `b` as an instance of `B`, but rather an instance of `A`. In order to model this case more precisely, we support `self` type in the annotation. The method me can now have the type, `() -> self`. In `the Kernel` module in `base_types.rb`, clone method also has the same type.

```
##% clone: () -> self
def clone() end
```

5.6 Parametric Polymorphism

To give precise types to container classes, DRuby supports *parametric polymorphism*, also called generics in Java. The following code show a segment of the `Array` class, which is parameterized by a *type variable* `t`, which is the type of the contents of the array:

```
##% Array<t>
class Array
  ##% at : (Fixnum) -> t
  def at(p0); end
  ##% first : () -> t
  ##% first : (Fixnum) -> Array<t>
  def first(p0 = 0); end
  ...
  ##% collect<u> : () {t -> u} -> Array<u>
  def collect() end
  ...
end
```

As usual, type variables bound at the top of a class can be used anywhere inside that class. For example, the type of the `at` method, which takes an index and returns the element at that index. Type variables can also be used in intersection types, e.g., the first method, takes either no argument, in which case it returns

the first element of the array or a value n, in which case it returns the first n elements of the array, which is itself an array of type `Array<t>`.

We also support polymorphism on methods. Consider the `collect` method, which for any type `u` takes a code block from `t` to `u` and produces an array of `u`. The `concat` method, which takes an array of `u`'s and (non-destructively) appends it to the end of the current array, producing a new array whose elements are either `t`'s or `u`'s.

DRuby also allows bounded quantifiers. So far, all of the polymorphic variables have had no explicit bound, which defaults to giving them an upper bound of the Top type. Alternatively, an upper bound may be explicitly given for any polymorphic type variable (including self). For example,

```
class A
  ##% smart_clone<self> ; self <= A () -> self
  def smart_clone() ... end
```

Here, smart_clone can only be called by (sub-)classes that are subtypes of A.

5.7 Tuple Types

The Array<t> type describes homogeneous arrays, whose elements all consists of the same type. However, Ruby's dynamic typing allows programmers to also create heterogeneous ararys, in which each element may be a different type. This is especially common for returning multiple values from a method, and there is even special parallel assignment syntax for it. For example, the following code

```
def f(); [1, true] end
a, b = f
```

assigns `1` to `a` and `true` to `b`. If we were to type the return value of `f` as a homogeneous `Array`, the best we could do is make it `Array<Object>` or `Array<Fixnum or Boolean>`, with a corresponding loss of precision. Our solution is to introduce a tuple type $(t_1, ..., t_n)$ that represents an array whose element types are, left to right, t_1 through t_n. When we access an element of a tuple using parallel assignment, we then know precisely the element type. (When inferencing types, DRuby might assign something a tuple type, but then there is a subsequent operation causes a loss of precision, such as mutating a random element or appending an array. In this case, DRuby falls back and treat it as an array, not a tuple anymore.)

You can annotate the example above as the following:

```
##% f : () -> (Fixnum, Boolean)
def f(); [1, true] end
a, b = f
```

5.8 First Class Methods

DRuby includes support for another special kind of array: method parameter lists. Ruby's syntax permits at most one code block (higher-order function) to be passed to a method. For example, the following code illustrates how a block argument of `collect` method can increment each element in the array by 1.

```
[1, 2, 3].collect {|x| x + 1 } # returns [2, 3, 4]
```

If we want to pass multiple code blocks into a method or do other higher-order programming (e.g., store a code block in a data structure), we need to convert it to a `Proc` object:

```
f = Proc.new {|x| x + 1}
f.call(3) # returns 4
```

A `Proc` object can be constructed from any code block and may be called with any number of arguments. To support this special havior, in `base_types.rb`, we declare Proc as follows:

```
##% Proc<^args,ret>
class Proc
  ##% initialize: () {(^args) -> ret} -> Proc<^args, ret>
  def initialize(); end
  ##% call : (^args) -> ret
  def call(p0); end
end
```

The `Proc` class is parameterized by a parameter list type `^args` and a return type `ret`. The `^` character acts as a type constructor allowing parameter list types to appear as first class types. The `initialize` method (the constructor called when `Proc.new` is invoked) takes a block with parameter types `^args` and a return type `ret` and returns a corresponding `Proc`. The call method takes then has the same parameter and return types.

As another example use of `^`, consider the `Hash` class:

```
##% Hash<k, v>
class Hash
  ##% initialize : () {(Hash<k,v>,k) -> v} -> Hash<k,v>
  def initialize(); end
  ##% default_proc : () -> Proc<^(Hash<k,v>,k),v>
  def default_proc(); end
end
```

The `Hash` class is parameterized by `k` and `v`, the types of the hash keys and values, respectively. When creating a hash table, the pro- grammer may supply a default function that is called when an ac- cessing a non-existent key. Thus the type of initialize includes a block that is passed the hash table (`Hash<k,v>`) and the missing key (`k`), and produces a value of type `v`. The programmer can later extract this method using the default proc method, which returns a `Proc` object with the same type as the block.

5.9 Subtype Relation

It is also possible to give a hint to DRuby that a certain class is a *subtype* of another. This is useful if you want to establish a relationship between classes so that a class is assumed to be another class. Consider again the example from Section5.1:

```
class A; end
class B < A; end
##% f : () -> Boolean
##% f : A -> Boolean
def f(x=A.new) true end
f(A.new) # type checks
f(B.new) # fails
```

As we mentioned above, DRuby cannot determine whether `B` is substitutable for A and therefore can not verify the latter call to f. This is because `B < A` declares a subclass relationship, which does not imply that

B is a subtype of A. The former only tells us that `B` inherits all the methods in `A`, and nothing about the type relationship between the two. B is still free to overwrite these methods with a type-incompatible version.

On the other hand, subtype annotation requires B to be a subtype of A and therefore redefing A's methods must preserve their type. This annotation also does one more thing: it allows DRuby to assume from the start of inference that B is a subtype of A since this will be later verified by the constraint solver. Therefore, our example is now well-typed with this annotation:

```
class A; end
##% B <= A
class B < A; end
##% f : () -> Boolean
##% f : A -> Boolean
def f(x=A.new) true end
f(A.new) # type checks
f(B.new) # fails
```

Thus when attempting to select a part of an intersection type, DRuby looks for either an exact name match, or a name that has annotationed as a subtype of the name required (note: this relation is transitive).

This annotation can also be useful to *verify* that a class is a subtype of another for documentation purposes. Recall that the subclass relationship does not constrain the types of overriding methods with respect to the types of inherited methods. Consider this valid code:

```
class A; def g() 0 end end
class B < A; def g() "1" end end
```

This is a legitimate program although the overriding `B#g` returns a `String` whereas the overriden method `A#g` returns a `Fixnum`. However, sometimes it is desired to enforce the subtype relation so that `B#g` must be a subtype of `A#g`. We can manually constrain this by introducing an annotation for subtype relation to `B` and its super type, `A`. Try runnig DRuby on the following code.

```
class A
  ##% g : () -> Fixnum
  def g() 0 end
end
##% B <= A
class B
  ##% g : () -> String
  def g() "1" end
end
```

Because of the presence of the subtype relation in the annotation, DRuby strictly checks for that relation between `B` and `A`. DRuby rejects this program due to the inconsistent subtype relation between `B#g` and `A#g` and generates this error message:

```
[ERROR] instance String used where Fixnum expected
         It does not support methods ~, |, zero?, truncate, to_int, times,
        step, singleton_method_added, rpower, round, remainder, rdiv, quo,
        prec_i, prec_f, prec, power!, numerator, nonzero?, nan?, modulo, lcm,
        integer, infinite?, id2name, gcdlcm, gcd, floor, finite?, downto,
        divmod, div, denominator, coerce, chr, ceil, abs, ^, >>, /, -@, -, +@,
        **, &
  in verifying declared subtype: B <= A
  in creating/updating type ::B
  at ./subtyperelation.rb:6
  in class definition ::B
  at ./subtyperelation.rb:6
  in annotation for g
  at ./subtyperelation.rb:8
```

6 Type System

6.1 Static Type Inference and Checking

The main feature of DRuby is static type analysis, which consists of type inference and type checking. It attempts to give a type to every object in a program and report any inconsistencies observed during the process (type inference), and checks the validity of type annotations against the inferred types and/or other type annotations (type checking). There are three kinds of errors DRuby reports that are critical to type analysis—*methods not supported*, *wrong arity to function*, and *subtype relation failure*.

6.1.1 Methods Not Supported

This will be probably the most common error you will have when using DRuby. It is a simple type error, where expected methods are absent from an object. For example,

```
class A; def f(); end end
A.new.g # there is no g in A
```

causes *method not supported* error because method `g` was invoked on `A.new`, but `A` has no method `g`. The error message that DRuby generate is apparent for this example. There is no short description, yet the long description is informative.

```
[ERROR] instance A does not support methods g
  in typing method call g
  at ./method.rb:2
  in typing ::A.new
  at ./method.rb:2
```

Let's consider another example where there is a type annotation for a method.

```
class A; def f() 0 end; def h() 2 end end
class B; def g() 1 end end
##% foo : A -> Fixnum
def foo(a) 3 end
foo(B.new) # A is expected
```

The method `foo` takes an `A` and returns a `Fixnum`, but `foo` was called with `B.new` at the end. Therefore, this is obviously a type error, and DRuby generates an error message as follows:

```
[ERROR] instance B used where A expected
        It does not support methods h, f
  in typing method call foo
  at ./method2.rb:5
  in typing ::B.new
  at ./method2.rb:5
```

The short description (the top most sentence) says, "`instance B used where A expected`", mentioning the nominal type `A`. This indicates that DRuby has taken into consideration the types presented by the annotation since the error message refers to the *expected* object as "`instance A`", not as some arbitrary type that has methods `h` and `f`. Therefore, it outputs more user-friendly description of the error. The long description still provide the details of the error which is the list of methods required by `A` but missing in `B` which are `f` and `h`. To illustrate this point further, consider this code

```
class A; def f() 0 end; def h() 2 end end
class B; def g() 1 end; def f() 0 end; def h() 2 end end
##% foo : A -> Fixnum
def foo(a) end
foo(B.new) # A is expected
```

where B now has `f` and `h` in addition to `g`. DRuby now accepts the program although `B.new` is not really an instance of `A`, because DRuby uses structural typing. Although the type annotation notifies DRuby that `foo` accepts an instance of `A`, DRuby interprets it as "`foo` accepts an instance of a class that has all the methods that `A` has (with appropriate types)".

6.1.2 Wrong Arity To Function

This is another common mistake that Ruby programmers often make—wrong number of arguments. If the number of actual parameters in a method call does not match with the number of formal parameters of the corresponding method, it is obviously an error. The following code

```
class A; def f(a) a.to_s end end
A.new.f()
```

defines method `f` in class `A`, but calls it with no argument. DRuby outputs as follows.

```
[ERROR] wrong arity to function, got no arguments, expected exactly 1 arguments
  in solving method: f
  in closed solving instance A <= ?
  in typing method call f
  at ./arity.rb:2
```

This kind of error is actually more common with formal parameters of a block argument. Consider the following code:

```
class A; def f(a, b) yield([a, b]) end end
A.new.f(1, 2) {|x, y| x * y}
```

Method `f` takes two arguments (`a` and `b`) and pass them as a single array to `yield`, but the block argument to `f` has two formal parameters `x` and `y`. Unfortunately, Ruby is lenient on the formal parameters in a block argument, so this mistake is difficult to notice. This is a bad programming style because it makes hard for programmers to detect real errors caused by the same programming habit. Therefore, DRuby reports this as an error. The correct coding style for the above example is

```
class A; def f(a, b) yield([a, b]) end end
A.new.f(1, 2) {|(x, y)| x * y}
```

where the tuple type `(x,y)` makes up a single type. This may seem unnatural at first, but it might make more sense when compare it to a slightly altered version of the example:

```
class A; def f(a, b, c) yield([a, b], c) end end
A.new.f(1, 2, 3) {|(x, y), z| x * y + z}
```

In this case, even Ruby will complain if `x` and `y` are not a tuple.

6.1.3 Subtype Relation Failure

When DRuby is unable to determine whether or not a given type is consistent with respect to an intersection type, it reports a subtype relation failure. For example, the following code

```
##% f : Symbol -> Boolean
##% f : String -> Boolean
def f(p0) true end
foo(1)
```

defines method `f` and asserts the type of the method as an intersection type. However, `f` was used with a `Fixnum`, causing an inconsistency with repect to the annotation. In fact, this problem occurs more often than you might expect. This is because if DRuby cannot determine whether or not the type of an actual argument is a subtype of the either Symbol or String, it considers it as a subtype relation failure. You can avoid this problem by adding subtype relation in type annotation. For more information on this topic, refer to Type Annotation section.

6.1.4 Other Errors

You might occasionally encounter other kinds of errors such as undefined constants, unresolve scopes, and etc. These are not critical to the analysis yet worth mentioning:

- Scoping errors - these errors usually happen when DRuby cannot find a particular class or module that it's looking for. Try to use specific paths or include the paths into RUBYLIB environment variable.
- Aliasing errors - if the source method of the aliasing operation cannot be found, DRuby warns the user.
- [TODO: Hmm...I'm actually not so sure how to categorize these]

7 Profile-Guided Analysis

NOTE: This feature is currently experimental

Ruby has dynamic language constructs that allow programmers to execute dynamic code and perform reflections on objects at run time. For instance, `eval` takes a string argument and executes the code on-the-fly. The following code snippet (excerpted from *text-highlight-1.0.2*)

```
ATTRIBUTES.each do |attr|
  code = <<-EODEF
  def #{attr}(&blk)
    color("#{attr}", &blk)
  end
  EODEF
  eval code
end
```

iterates through `ATTRIBUTES`, an array of strings. For each element it creates a string code containing a new method definition, and then evaluates code. The result is certainly elegant—methods are generated dynamically based on the contents of an array. However, no reasonable static type system will be able to analyze this code.

Actually, even `require` behaves dynamically. Unlike the `include` keyword in C, which is actually handled by a preprocessor, Ruby's `require` works as a method call which means that Ruby dynamically reads the file, and evaluates the contents as a string code on-the-fly. The argument to the `require` call, which is the path to a target file, can be a non-literal expression whose value might not be statically determined. For instance, `require File.dirname(__FILE__) + '/../lib/mylib'` takes an expression rather than a string literal in order to resolve the path with respect to the current file's path.

Instead of performing a complex dataflow analysis such as string literal analysis, we take a hybrid approach. In order to determine how these dynamic features are used, DRuby can first run the program to gather profiles of their use. Since many Ruby programs are developed with a test suite with (hopefully) good code coverage, running these tests provides exactly the information we need. Once we have executed the test suite, DRuby can then improve its static analysis based on the information it has gathered, thereby improving the precision of its analysis.

You can run the profile-guided analysis by running the following command:

```
$ druby --dr-profile filename
```

It is also common among Ruby programmers to use Rake to drive their testsuites. In this case, you should specify *filename* as the rake executable: *`PATH_TO_RAKE_BIN`*`/rake test`, along with `--dr-profile` option, which will allow DRuby to observe the tests described by your Rakefile. If the program comes with a custom testing framework, you must run its own test runner. It is also possible to run the program directly and not the testsuite. This should also work fairly well in practice because we learned that most dynamic behaviors occur during the load time. However, it is recommended that you would write test cases that has a reasonable path coverage.

So what is really happening here with the profiling option? Consider the *text-highlight* example again. When we run druby with the profiling option, DRuby instruments the `eval` method call and executes the program using the Ruby intepreter. This instrumentation code, at run time, records the arguments to the `eval`. Once the profiling phase is complete, DRuby reads in the profiled data and transforms the `eval` method call into a case statement as follows.

```
ATTRIBUTES.each do |attr|
  code = <<-EODEF
  def #{attr}(&blk)
    color("#{attr}", &blk)
  end
  EODEF
  case eval_arg
  when "def none(&blk) ... end"
    def non(&blk) ... end
  when "def reset(&blk) ... end"
    def reset(&blk) ... end
  else
    raise Error
  end
end
```

Now, because all possible cases of the `eval` call are replaced with actual code, simulating the run time behavior. This transformation allows DRuby to analyze this code now. Note that the transformed code aborts the program if an unseen string is passed to eval. This is obviously not ideal and future versions of DRuby will be able to add extra dynamic checks at this point so that unseen strings can still be safely evaluated.

This profiling option is crucial to analyzing programs that use the Ruby standard library, since it makes heavy use of dynamic features. Consequently, many false positives are inevitable without this profiling technique. In future versions of DRuby, a snapshot of pre-profiled data will be provided with the distribution, so the profiling phase is not required for every time DRuby is run.

Here is the complete list of the dynamic language constructs that DRuby currently handles:

- `require`, `load`,
- `eval`, `instance_eval`, `class_eval`, `module_eval`,
- `send`, `__send__`,
- `instance_variable_get`, `instance_variable_set`, `class_variable_get`, `class_variable_set`, `const_get`, `const_set`
- `method_missing`.

8 Ruby Intermediate Language (RIL)

The DRuby implementation includes Ruby Intermediate Language (RIL). In order to analyze Ruby source code, we first parse the source code into a raw AST, and then translate and simplify that tree into RIL for further analysis. For example, the type analysis is done on top of RIL, not the actual source code. Code instrumentation and transformation for the profiling process are also done within RIL. DRuby is designed so that it is easy to perform various operations such as exploring, instrumenting, transforming, and analyzing on RIL. More detailed documentation on RIL will be provided in future distribution. Meanwhile, here is a short example that parses code and transforms every method call to `foo` into a call to `bar`.

```
 1 open Cfg
 2 open Visitor
 3 class foo_visitor =
 4 object(self)
 5   inherit default_visitor
 6   method visit_stmt s = match s.snode with
 7     | MethodCall(lhs_o, mc) ->
 8         begin match mc.mc_msg with
 9           | `ID_MethodName("foo") ->
10               let new_mc = {mc with mc_msg = `ID_MethodName("bar")} in
11               let new_snode = MethodCall(lhs_o, new_mc) in
12               let new_stmt = mkstmt new_snode s.pos in
13                 ChangeTo new_stmt
14           | _ -> DoChildren
15         end
16     | _ -> DoChildren
17 end
18 let _ =
19   let fname = Sys.argv.(1) in
20   let ast = Parse_helper.parse_file fname in
21   let cfg = Cfg_refactor.refactor_ast ast in
22   let visitor = new foo_visitor in
23   let new_cfg = visit_stmt visitor cfg in
24     Cfg_printer.CodePrinter.print_stmt stdout new_cfg
```

First, we are going to open `Cfg` and `Visitor` modules so that functions and values in the modules can be accessed without the namespaces. Lines 3-17 illustrates how a CFG visitor class can be defined to manipulate control flow graph. Refer to `src/cfg/cfg.mli` and `src/visitor.mli` for more information on the CFG module and the Visitor class. We override the method `visit_stmt` so that whenever it encounters a method call to "`foo`", it changes the node to a method call to "`bar`". For other CFG nodes, it keeps visiting the current node's children.

The main function parses the specified file and refactors the AST into a CFG. Then, this CFG is transformed by the visitor (at line 22 and 23). Finally, it prints the modified source code into standard output.

9 Contact

If you questions about using DRuby or about the code, please contact Mike Furr <furr@cs.umd.edu>