Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Στατική ανάλυση για έλεγχο λαθών στη γλώσσα Ruby

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ ΒΑΘΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2012

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Στατική ανάλυση για έλεγχο λαθών στη γλώσσα Ruby

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΙΚΟΛΑΟΣ ΒΑΘΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Νοεμβρίου 2012.

Αθήνα, Νοέμβριος 2012

....

Νικόλαος Βάθης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο προγραμματισμός ηλεκτρονικών υπολογιστών είναι μία επιστήμη που έχει γνωρίσει ραγδαία ανάπτυξη τις τελευταίες δεκαετίες. Τη σημερινή μέρα είναι εύκολο για τον οποιονδήποτε να γράψει κάποιο πρόγραμμα σε κάποια γλώσσα προγραμματισμού. Αντίθετα, οι τεχνικές απόδειξης ότι ένα πρόγραμμα όντως κάνει αυτό που θέλει ο προγραμματιστής δεν είναι εξίσου ανεπτυγμένες. Η πιο απλή μορφή απόδειξης για τη σωστή λειτουργία ενός προγράμματος είναι ο έλεγχος ορθότητας από ένα σύστημα τύπων. Σε αυτή τη διπλωματική κατασκευάζουμε ένα στατικό σύστημα τύπων που εντοπίζει σφάλματα σε αντικειμενοστρεφείς γλώσσες προγραμματισμού με δυναμικό σύστημα τύπων. Στη συνέχεια, υλοποιούμε αυτό το σύστημα τύπων για ένα υποσύνολο της γλώσσας προγραμματισμού Ruby, το οποίο εν συνεχεία επεκτείνουμε ώστε να καλύψει μεγαλύτερο μέρος της γλώσσας Ruby.

Το σύστημα τύπων που κατασκευάζουμε είναι βασισμένο στα Success Typings, τη θεωρία πίσω από το εργαλείο Dialyzer που βρίσκει σφάλματα τύπου σε προγράμματα της γλώσσας Erlang. Η βασική του ιδιότητα είναι ότι δεν υπάρχει περίπτωση να εντοπίσει σφάλμα σε πρόγραμμα το οποίο δεν έχει σφάλμα. Δηλαδή, εντοπίζει μόνο υπαρκτά σφάλματα. Το εργαλείο που εντοπίζει τα σφάλματα στο υποσύνολο της γλώσσας Ruby είναι εύκολο στη χρήση του, καθώς δεν χρειάζεται καμία υποσημείωση τύπων για να λειτουργήσει σωστά.

Τέλος, παρουσιάζουμε μία βήμα προς βήμα σύγκριση με τη Diamondback Ruby, τον κύριο ανταγωνιστή στον εντοπισμό σφαλμάτων τύπου στη γλώσσα Ruby, σχολιάζοντας τις εξόδους των δύο εργαλείων.

Λέξεις κλειδιά

Στατική ανάλυση, έλεγχος τύπων, εντοπισμός σφαλμάτων, σύστημα τύπων, Ruby

Abstract

Computer programming is a very widespread science, practiced by many individuals. Nowadays, it is easy for almost anyone to write a computer program. Unfortunately, it is still very difficult to prove that a computer program will behave as it is supposed to, when executed. The most basic form of such a proof is type checking. So, in this thesis, we present a static type system that detects type errors in dynamically typed object oriented languages. Also, we implement this type system for a subset of the Ruby programming language, which we then try to extend to cover a bigger part of Ruby.

The type system that we present is based on Success Typings, the theory behind the Dialyzer, a tool that detects type errors in the Erlang programming language. Its most interesting property is that it detects only definite errors, and it will never detect a false positive. The tool that we propose should be easy to use, as it requires no intervention of the programmer, as for example in the form of type annotations.

Finally, we present a step by step comparison between our tool and Diamondback Ruby, our main competitor in the field of static type checking in Ruby, commenting on the outputs of the tools in the process.

Key words

Static analysis, type checking, error detection, type system, Ruby

Ευχαριστίες

Κατ’αρχάς, θα ήθελα να ευχαριστήσω τον καθηγητή μου, Νικόλαο Παπασπύρου, για όλα όσα με δίδαξε σε όλα τα χρόνια που σπουδάζω, και για την καθοδήγηση που μου προσέφερε ως επιβλέπων καθηγητής της διπλωματικής μου. Εν συνεχεία, θα ήθελα να ευχαριστήσω και τον καθηγητή μου Κωστή Σαγώνα για όλα όσα με δίδαξε. Θα ήθελα επίσης να ευχαριστήσω την οικογένειά μου για την υποστήριξη που μου παρείχε σε όλη τη διάρκεια της φοίτησής μου. Τέλος, θα ήθελα να ευχαριστήσω όλους μου τους φίλους, με τους οποίους αντιμετωπίσαμε μαζί τις δυσκολίες της σχολής.

Νικόλαος Βάθης,

Αθήνα, 23η Νοεμβρίου 2012

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-12, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2012.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

This thesis aims at defining a type system suitable for error detection in dynamically typed object-oriented programming languages. It also aims at implementing the type system, in the form of a static analysis tool for a subset of the Ruby programming language. The type system is based on Success Typings, the theory behind the Dialyzer, a tool that performs static analysis and error detection in the Erlang programming language, and also a part of the Erlang standard implementation [Dial].

The great debate is whether to aim for soundness or completeness. This type system favors soundness, so that every error it detects is an actual error, and cannot be a false positive.

1.2 Motivation

Programming is a very widespread practice today and programming languages are a critical part of it. From simple tasks to mission critical applications, computers can do almost every non-trivial task efficiently. While programming can be practiced by anyone, making sure that a program won't go wrong is a quite cumbersome task.

In mission critical applications, it is mandatory to prove the correctness of a system before it enters production. This process is usually lengthy and involves quite complex procedures. On the other hand, when facing a simpler task, validation is still desired, but it has to be provided by very fast means.

On those grounds, type checking is the most basic type of program validation. By giving each program variable a type, minimal proofs can be constructed that a program won't result in some specific errors. For example, a tool can determine very fast that adding a string to an integer will result in a crash. Or worse, if the language is close to machine language, this addition will result in some meaningless number. By type checking a program, it can be guaranteed that some mistakes will never happen.

The type system is a core part of a programming language. It provides a bridge between mathematics and computer programming, is part of the documentation of program components, and provides guarantees of the correctness of a program. There are many approaches to the problem of type checking. Some languages use static type systems, by validating all the types while the program is being compiled, whereas other languages use dynamic type systems, and perform run time checks.

Although most widespread programming languages impose static type systems, dynamically typed languages have a reasonable market share. On the one hand, dynamic typing solves the issues of polymorphism in an elegant way, as there is no way a perfectly valid function call will fail due to the strictness of the type system. On the other hand, dynamically typed languages have no guarantee that every program that the parser accepts is actually a valid program, at least not before the actual excecution. Also, it is usually difficult to find the faulty code, as the error occurs later in the execution.

Over the years, many theories have emerged on how the safety of static typing and the expressivity of dynamic typing can be gracefully combined. To name a few, a theory can impose a strict type system, which rejects valid programs, it can use runtime profiling to extract static typing information or it can use some type of annotation language. As a matter of fact, all of these methods have been tried with Ruby [Furr09, An11]. While all methods are acceptable to some people, they are highly unacceptable to others. So, the question arises, if there actually is a widely acceptable way to impose a static type system on a dynamically typed language, so that it saves precious time from a programmer, without imposing an unacceptable overhead. Success typings seem to have all these qualities, as they aim at definite error detection, and not at type safety. After all, the truth is that a programmer will be reluctant to adopt a static type checker, unless this type checker is fast and does not reject most of his programs that run flawlessly. After all, he most probably chose a dynamically typed language in order to do rapid prototyping, and to exploit the extra expressivity, and will not easily give these abilities back again.

Finally, it should be noted that most dynamically typed languages are scripting languages, and favor test driven development, which is presented as an alternative to static type checking. It is highly debatable whether test driven development can actually replace type checking, as type errors usually happen by mistake, where least expected, and test driven development is based upon having a minimal knowledge of where the mistakes might occur. Nevertheless, if a programmer can have both static analysis tools and unit tests during development, at no extra cost, he might as well use both.

This thesis presents a type system, in which every valid program type checks, at the expense of also accepting some programs that will crash when executed. By implementing a tool that is based on this type system, the result should be an easy to use tool, which only rejects programs that have definite errors, and should not have any false positive whatsoever. This way, this tool can either be used with no extra overhead, or be possibly extended to use an annotation language.

1.3 Outline of the Thesis

The rest of the thesis is organised as follows: First of all, Chapter 2 gives an introduction to dynamic typing, and the Ruby programming language. Chapter 3 introduces the subset of Ruby that we will be using in order to introduce our type system. Chapter 4 introduces the idea of success typings, and shows how they can fit in Core Ruby. In Chapter 5 we present the rules for the generation of success typings for Core Ruby. In Chapter 6 we present the comparison between our tool and Diamondback Ruby. And finally, in Chapter 7, we reach our conclusion, and present some future work that can be done on the subject.

Chapter 2

Dynamic Typing and Ruby

2.1 Introduction to Dynamic Typing

A dynamically typed language is a programming language that performs type checks while the program executes, and not before or during the compilation. Some examples of dynamically typed languages are Python, Ruby, Erlang, PHP and JavaScript. Dynamically typed programming languages favor more expressivity, metaprogramming, and polymorphism without complex additions to the type system, such as templates or typeclasses.

As it is easier to explain by example, below lies a C-like pseudocode in a dynamically typed language:

```
x = 5;
x = x + 1;
x = "hello";
printf("%s\n", x);
```

Listing 2.1: Dynamically typed pseudocode

Here, the variable x is first assigned an integer value, and then, in the middle of the program, is assigned a string. No declaration is ever needed, and the compiler does not try to guess the type of each variable.

There are many challenges in finding type errors in a dynamic type system without executing the program. For example, in most dynamically typed languages, a variable is allowed to change types throughout the execution. This is true for Ruby, Python, PHP and JavaScript, but not for Erlang, which allows only a single assignment to each variable. Also, the variables in most object oriented languages are references to variables, and not variables themselves, so extreme caution must be taken for the possible aliasing of variables. And last, but not least, most, if not all dynamically typed languages, have a way to express higher order functions, which adds more complexity to a static type checker.

2.2 The Ruby Programming Language

2.2.1 Introduction to Ruby

The Ruby programming language is a scripting object oriented language, with dynamic typing. It was designed with the programmer in mind, so its syntax is very high level. As a consequence, it is not easy to parse with a tool. To add to that, there exists no up to date standard for the language, except for the official implementation, and perhaps the capability to run the Ruby on Rails framework. Nevertheless, it is strongly typed, and no type coersions happen implicitly. The last property makes it an interesting candidate for a proof of concept tool that analyzes it statically.

So, lets just see how one defines a class, creates an instance, and calls a method:

```ruby
class Foo
  def fun
    ”Hello world”
  end
end

x = Foo.new
puts x.fun
```

Listing 2.2: Ruby object orientation

A more interesting example follows, where the dynamic typing can be seen in action:

```ruby
def printmult x,y
  puts x * y
end

printmult 6, 7
printmult ’*’, 42
```

Listing 2.3: Ruby dynamic typing

So, this function multiplies its arguments and prints the result. The first call seems obvious. For the second one, a certain background is needed. Every operator is actually a method, applied to the first argument. So, in order for the second call to not result in an error, the String class must implement the * method. Luckily, it does so, and the second call will print forty two stars.

Should the printadd function be called with an instance of Foo, defined by the previous example, and an integer, though, the program will crash, saying that class Foo does not implement the * method.

```ruby
 1 class Foo
 2   def fun
 3     ”Hello world”
 4   end
 5 end
 6
 7 def printmult x,y
 8   puts x * y
 9 end
10
11 x = Foo.new
12
13 printmult x, 6  # this call will crash
```

Listing 2.4: Ruby dynamic typing error

An interesting fact is that the execution will crash at line 8, probably the least faulty line of the whole program. It is actually difficult to blame the erroneous code. There are two possibilities in this example. Either the error is at line 13, and the programmer did not mean to call this function, or the error lies at line 11, where the variable x was initialized with the wrong type.

So, dynamic type checking is useless without a proper stack trace, and even then, it is far from perfect in blaming the erroneous code. One final example, where it is more obvious that the function call is not the true culprit:

```ruby
intarray = [1,2,3]
intarray.push 4
intarray.push ”a”  # oops
intarray.push 6
```

```ruby

for x in intarray do
  puts x + 1
end
```

Listing 2.5: Ruby dynamic typing error 2

From the name of `intarray`, it seems rather obvious that the programmer did not intend to have a string in this array. Yet the execution ended up having one, and crashing inside the `for` loop. Of course, such an addition to the array might not be that obvious as in this example.

So, to sum up these examples:

- Ruby is object oriented and dynamically typed,
- The main reason for a Ruby program to crash when executed, is that a class does not implement the method being called,
- It is difficult to blame the erroneous code by automatic means.

2.2.2 Type checking Ruby

As seen in the previous section, our best bet in statically type checking Ruby is to check whether all method calls correspond to an instance method. But that is not the only type error found in Ruby. Should a string be added to an integer, the Ruby interpreter will complain that it cannot convert the argument to string. So, the Ruby built-in functions are statically typed. Except for that, the behaviour when a method is missing can be changed, by overriding the “`method_missing`” method of a class.

Also, Ruby supports metaprogramming. To be more specific, Ruby allows method definitions that are not resolved at compile time. So, a static tool might have to actually execute Ruby code before performing a type check. An example:

```ruby
class Color
  Colors = [ ["red",[1,0,0]], ["green",[0,1,0]] ]

  Colors.each { |t| define_method t[0] { t[1] } }
  # now class Color has methods red and green
end
```

Listing 2.6: Ruby metaprogramming

To make matters even worse, with a little bit of effort, and maybe some `eval` trickery, someone can add and remove methods from a class at run time. So, some clever programmer can render any static analysis tool useless, with little to no effort.

```ruby
class Foo
end

def genm x, y
  eval "class Foo; def #{x}; #{y}; end; end"
end

x = Foo.new

genm "six", 6
genm "seven", 7

```

```
puts x.six * x.seven # prints 42
```

Listing 2.7: Ruby dynamic metaprogramming

So, every static type checking tool must depend on some programmer and tool implicit contract. If the programmer tries to outsmart the tool, then the tool will simply fail.

It seems that the only solution to type checking Ruby is to begin bottom up, by creating a minimal language, adding features, and see which features are problematic and which are not. Let’s see how such a language could look like.

Chapter 3

Core Ruby

3.1 Syntax

The abstract syntax of Core Ruby should be fairly simple, yet allow for much expressivity. It should be Turing Complete, and resemble Ruby, without having excess Rubyisms, in order to be easy to adapt to other object oriented languages. So, it should have at least classes, methods, fields. Core Ruby should not be free of side effects, in order to resemble most object oriented languages. Also, a branching mechanism is mandatory.

Luckily, a very good language can be retrieved from [An11], and by performing some small changes, we end up with Core Ruby, which can be seen on Figure 3.1.

In order to maintain some Ruby compatibility, the allowed names were chosen in the following way:

- local variable names begin with a small letter
- field names begin with an @ symbol
- method names begin with a small letter, or consist of a single mathematical operation symbol.
- class names begin with a capital letter

3.2 Operational Semantics

Since a language definition is not complete without proper semantics, the operational semantics of Core Ruby can be seen in Figures 3.2 and 3.3, while the definition of each symbol can be seen in Figure 3.4.

Each rule consists of six elements. The expression being evaluated alongside the memory state, the class environment in which it is being evaluated (the object for which the current method has been called), the list of classes, which stays the same for the entire execution, and the resulting expression and memory state.

3.3 Examples

So, lets write a simple example in this language:

$$
\begin{array}{rrllll}
\textit{expressions} & e ::= & \texttt{nil} & \mid\ \texttt{self} & \mid\ x & \mid\ @f \\
 & \mid & x = e & \mid\ @f = e & \mid\ A.\texttt{new}(e_1, ..., e_n) & \mid\ e.m(e_1, ..., e_n) \\
 & \mid & e; e' & \mid\ (e) & \mid\ \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e''\ \texttt{end} & \\
\textit{methods} & m ::= & \texttt{def}\ m(x_0, ..., x_n)\ e\ \texttt{end} & & & \\
\textit{classes} & c ::= & \texttt{class}\ A\ d*\ \texttt{end} & & & \\
\textit{programs} & P ::= & c*\ e & & &
\end{array}
$$

x : local variable names	$@f$: field names
m : method names	A : class names

Figure 3.1: Core Ruby Syntax

$$
\frac{(e_1, \mu) \overset{l,c*}{\rightarrow} (e_1', \mu')}{(e_1;\ e_2, \mu) \overset{l,c*}{\rightarrow} (e_1';\ e_2, \mu')} \qquad (v;\ e, \mu) \overset{l,c*}{\rightarrow} (e, \mu)
$$

$$
\frac{(e_{obj}, \mu) \overset{l,c*}{\rightarrow} (e_{obj}', \mu')}{(e_{obj}.\texttt{m}(e_0, ..., e_n), \mu) \overset{l,c*}{\rightarrow} (e_{obj}'.\texttt{m}(e_0, ..., e_n), \mu')}
$$

$$
\frac{(e_i, \mu) \overset{l,c*}{\rightarrow} (e_i', \mu')}{(v_{obj}.\texttt{m}(v_0, ..., v_{i-1}, e_i, e_{i+1}, ..., e_n), \mu) \overset{l,c*}{\rightarrow} (v_{obj}.\texttt{m}(v,0, ..., v_{i-1}, e_i', e_{i+1}, ..., e_n), \mu')}
$$

$$
\frac{(e, \mu) \overset{l,c*}{\rightarrow} (e', \mu')}{(x = e, \mu) \overset{l,c*}{\rightarrow} (x = e', \mu')} \qquad \frac{(e, \mu) \overset{l,c*}{\rightarrow} (e', \mu')}{(@f = e, \mu) \overset{l,c*}{\rightarrow} (@f = e', \mu')}
$$

$$
\frac{(e_1, \mu) \overset{l,c*}{\rightarrow} (e_1', \mu')}{(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \mu) \overset{l,c*}{\rightarrow} (\texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \mu')}
$$

$$
\frac{v \not\equiv nil}{(\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \mu) \overset{l,c*}{\rightarrow} (e_1, \mu')} \qquad (\texttt{if}\ nil\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \mu) \overset{l,c*}{\rightarrow} (e_2, \mu')
$$

Figure 3.2: Core Ruby Operational Semantics

$$(self, \mu) \overset{l,c*}{\to} (l, \mu)$$

$$\frac{\mu = \langle h, s\rangle \quad (x \mapsto v) \in s}{(x, \mu) \overset{l,c*}{\to} (v, \mu)} \qquad \frac{\mu = \langle h, s\rangle \quad \mu' = \langle h, s\{x \mapsto v\}\rangle}{(x = v, \mu) \overset{l,c*}{\to} (v, \mu')}$$

$$\frac{\mu = \langle h, s\rangle \quad (l \mapsto \langle A, d\rangle) \in h \quad (@f \mapsto v) \in d}{(@f, \mu) \overset{l,c*}{\to} (v, \mu)}$$

$$\frac{\mu = \langle h, s\rangle \quad (l \mapsto \langle A, d\rangle) \in h \quad \mu' = \langle h\{l \mapsto \langle A, d\{@f \mapsto v\}\rangle\}, s\rangle}{(@f = v, \mu) \overset{l,c*}{\to} (v, \mu')}$$

$$\frac{\mu = \langle h, s\rangle \quad l' \notin dom(h) \quad \mu' = \langle h\{l' \mapsto \langle A, \emptyset\rangle\}, s\rangle}{(A.\texttt{new}(e_0, ..., e_n), \mu) \overset{l,c*}{\to} (l'.\texttt{initialize}(e_0, ..., e_n);\ l', \mu')}$$

$$(with\ \langle l', s'\rangle\ do\ v, \mu) \overset{l,c*}{\to} (v, \mu)$$

$$\frac{(e, \langle h, s'\rangle) \overset{l',c*}{\to} (e', \langle h', s''\rangle)}{(with\ \langle l', s'\rangle\ do\ e, \langle h, s\rangle) \overset{l,c*}{\to} (with\ \langle l', s''\rangle\ do\ e', \langle h', s\rangle)}$$

$$\frac{\mu = \langle h, s\rangle \quad (l' \mapsto \langle A, d\rangle) \in h \quad (\texttt{def}\ m(x_0, ..., x_n) = e) \in d^* \quad (\texttt{class}\ A = d^*) \in c^*}{(l'.\texttt{m}(v_0, ..., v_n), \mu) \overset{l,c*}{\to} (with\ \langle l', \{x_0 \mapsto v_0, ..., x_n \mapsto v_n\}\rangle\ do\ e, \mu)}$$

Figure 3.3: Core Ruby Operational Semantics (Continued)

$$\begin{aligned}
\mu &::= \langle h, s\rangle \\
s &::= \ \emptyset \mid s,\ x \mapsto v \\
h &::= \ \emptyset \mid h,\ l \mapsto \langle A, d\rangle \\
d &::= \ \emptyset \mid d,\ @f \mapsto v \\
v &::= \ nil \mid l
\end{aligned}$$

μ : memory	h : heap	s : stack
x : local variable	v : value	l : object reference
A : class name	d : field list	$@f$: field

Figure 3.4: Core Ruby Operational Semantics Symbols

```
class Bar
  def bar(x)
    nil
  end
end

class Foo
  def foo(x,y)
    x.bar(y)
  end
end

foo = Foo.new;
foo.foo(Bar.new, Bar.new)
```

Listing 3.1: Core Ruby simple example

In order to write examples that have any meaning, Core Ruby needs arithmetic. Since the language is Turing Complete, there are many ways to do that. But, considering that `if` expressions rely on whether the condition is nil or anything else, it is fairly logical that the bits will be represented by fields that are either nil or something else. Since there are only two literals in our language, nil and self, those will be used.

Even then, there are many options. One choice is to use fixed point arithmetic, by deploying a brute force increase method with eight if clauses, and define addition by increasing, and multiplication by adding. While this works, it is a rather slow implementation.

Instead, the decision was made to define infinite precision numbers represented with a singly linked list of bits. Also, the addition was implemented as an one bit adder with carry. This way arithmetics should have linear complexity instead of being exponential in relation to the size of the numbers.

So, a simple preprocessor should be coded, to convert natural number literals to Core Ruby syntax like this:

```
42 = Nat.new.x2(Nat.new.x2p1(Nat.new.x2(Nat.new.x2p1
                (Nat.new.x2(Nat.new.x2p1(Nat.new.zero(nil)))))))
```

This way, and by also adding code to the interpreter, so that it can convert this representation to natural literals, Core Ruby can do arithmetic without the slightest change in the semantics.

So, some examples that seem more real world:

```
class List
  def tabulate(tup) # { funobj, count }
    @data = tup.getFst(nil).fun(tup.getSnd(nil));
    if tup.getSnd(nil).iszero(nil) then
      @next = nil
    else
      @next = List.new;
      tup.setSnd(tup.getSnd(nil).-(1));
      @next.tabulate(tup)
    end;
    self
  end

  def foldl(tup) # { funobj, acc }
    if @next then
      accelem = Tuple.new;
      accelem.setFst(@data);
      accelem.setSnd(tup.getSnd(nil));

      tup.setSnd(tup.getFst(nil).fun(accelem));

      @next.foldl(tup)
    else
      tup.getSnd(nil)
    end
  end
end

class Double
  def fun(x)
    x.+(x)
  end
end

class Sum
  def fun(tup)
    tup.getSnd(nil).+(tup.getFst(nil))
  end
end

tup = Tuple.new;
tup.setFst(Double.new);
tup.setSnd(6);
lst = List.new.tabulate(tup);

tup = Tuple.new;
tup.setFst(Sum.new);
tup.setSnd(0);

lst.foldl(tup)
```

Listing 3.2: Core Ruby higher order example

This example creates a list by doubling all numbers ranging from 1 to 6, and then adds them by using a freshly created higher order function, similar to the foldl function most functional languages have (and similar to the Ruby `inject` method of the `Array` class). In order for this example to be complete, it should be noted that the `Tuple` class is trivial to write, as it only has two setters and two getters. The comments present at the function headers show what the arguments should contain.

Another fairly complex example:

```
class Tree
  def addLeft(left)
    @left = left
  end

  def addRight(right)
    @right = right
  end

  def setData(data)
    @data = data;
    self
  end

  def addRec(u)
    sum = 0;
    sum = sum.+(@data);
    if @left then sum = sum.+(@left.addRec(nil)) else nil end;
    if @right then sum = sum.+(@right.addRec(nil)) else nil end;
    sum
  end
end

a0 = Tree.new.setData(0);
a1 = Tree.new.setData(1);
a2 = Tree.new.setData(2);
a3 = Tree.new.setData(3);
a4 = Tree.new.setData(4);
a5 = Tree.new.setData(5);
a6 = Tree.new.setData(6);

a0.addLeft(a1);
a0.addRight(a1);

a1.addLeft(a2);
a1.addRight(a3);

a2.addLeft(a4);
a2.addRight(a5);

a3.addLeft(a6);

a0.addRec(nil)
```

Listing 3.3: Core Ruby trees

This example constructs a binary tree (actually it is a directed acyclic graph, but this should not really matter), so that the root has number 0 in it, while each subtree has all numbers from 1 to 6, once. Then, it adds all the numbers.

By now, it should be fairly obvious that the language is Turing Complete, although there is no formal proof, and also allows for a great deal of expressivity. Also, it has some features that make static type checking more difficult, such as the absence of built-in arrays. This feature seems rather obligatory, for two main reasons. First of all, it is not difficult to create an obscure class in plain Ruby that emulates the interface of an array (operators are method calls after all). Secondly, a Ruby array can have values of different types at each position. So, it is from very difficult to utterly wrong to treat an array as a built-in, from the perspective of the static type checker, unless the checker does some very pessimistic assumptions.

Chapter 4

Success Typings

4.1 Overview

So, what are success typings, and how do they fit in this thesis? The best way to introduce success typings is to introduce them intuitively, and try to add formal definitions later, where possible.

A success typing is a type information we assign to an expression, that gives information about an upper bound on the possible types this expression can have. So, if such a set can be computed, and if we can somehow later on determine the actual type of the expression, we can just see if the actual type exists in the success typing. If not, by definition, we have a type error.

This makes more sense in method arguments. We try to determine what types an argument can have, by examining the code of the method. Then, should we detect a method call to the aforementioned method, we compare the type of the actual parameter with the success typing of the formal parameter, and try to determine whether a type clash has occured.

4.2 Formal Definition

As stated in the introductory publication of the success typings [Lind06], a success typing of a function f is a type signature $(\overline{\alpha}) \to \beta$, such that whenever an application $f(p)$ reduces to a value v, then $v \in \beta$ and $p \in \overline{\alpha}$.

Since we are going to impose a type system to more than functions, a more generalized definition would be appropriate. So, a success typing of an expression e is a type signature t, such that when e reduces to a value v, then $v \in t$.

Note that a success typing does not guarantee type safety. It simply guarantees that, should the expression be type safe and reduce to a value, this value will have a type that belongs in its success typing. This way, should we observe in any way that the success typing of an expression is in conflict with an observed actual type, the evaluation of this expression cannot possibly succeed. So, for example, if we infer that a method argument has a success typing t, and later in the analysis that method is called with an argument of success type st, and $t \notin st$, we have detected a definite type error.

4.3 Dynamic Typing and Success Typings

The reason for trying to find only definite errors is fairly simple. Most programmers who favor a dynamically typed language will be hesitant to use a tool that limits the expressivity of the aforementioned language. If a tool finds a lot of errors that simply are not there, that tool will be used in very rare cases, such as in mission critical applications, or when big money is involved. But if a tool finds only definite errors, even if the number of errors it detects is rather small, and if this tool is easy to use and runs fast, there is no reason for someone to not use it, even out of disbelief for the tool itself.

After all, the problem is much worse in a dynamically typed object oriented language, as it is very easy to have false positives, and enrage the average programmer. First of all, the biggest part of the call graph is not known at compile time (and in the case of Core Ruby, none of it), and even with careful analysis, it is very likely to miss a large portion of it, or generate a wrong one. Add this to the extreme aliasing existing in all object oriented languages and it is quite easy to add a constraint that should not be there, by speculating and not by knowing. More on this later, when we try to compare our tool with existing implementations.

So, in order to infer the success typing of each expression, we are going to devise a constraint-based type inferencer. Many challenges have to be overcome. Success typings were conceived for the Erlang programming language, which has some properties that make type checking more precise, and somewhat easier.

Erlang allows a single assignment for each variable, the assignment being treated similarly to a Prolog unification. Variables can be either unbound or bound, and consecutive "assignments" actually unify their arguments, raising an error should the two terms differ. So, once bound, the type of a variable cannot change. Also, although Erlang has facilities for higher order functions, it has a wide range of primitives and built-ins that cannot be overriden (opposed to, for example, the square bracket operator in an array access in Ruby), that can be treated in some special, practical way that detects definite errors. Last, but not least, by default each function is called with static dispatch. Unlike Ruby, where everything is higher order, and every method call is dispatched dynamically. For example, from an Erlang `X + Y` expression, we can infer that `X` and `Y` are some kind of numbers. On Ruby, though, the only thing that we can infer is that `X` implements the `+` method, and if, and only if we know, for some reason, the type of `X`, we can try to see if the `Y` argument is an acceptable argument to this method.

When trying to solve such a practical problem, it is impossible to guarantee that some solution will work for every possible case. After all, such an "evil" example has been shown in the previous chapter (Listing 2.7), and it is fairly easy to create an "evil" code in Erlang that confuses the Dialyzer (e.g., Figure 4.1) to generate a false positive. It is far more important to guarantee that a tool works well for most cases, if the tool is to reach production. And the "most cases" are defined by the paradigms a language (and its community) favors, not by the paradigms a language can actually support, usually in some confusing way.

Just to clarify, it is not every case of `eval` that renders the tool useless. It is the use of `eval` that invalidates the typings of an expression. Simply put, any eval that changes the environment of execution, either by adding methods or by changing variables. Otherwise, the tool can safely ignore that eval and not miss the target of finding only definite, but not all, errors. It is highly debatable whether non-isolated eval invocations are considered good practice, in the average case.

4.4 Success Subtyping

Before finegraining the success typings theory to our language, there is a last detail. Since a success typing is not a classic typing, its subtyping is special. A success typing contains all the types an expression can have, so it actually captures a set of possible values for this expression. With this in mind, it is easy to define the success subtyping.

$x : a, y : b, a \sqsubseteq_S b$ denotes that the set of possible values of variable x is a subset of the set of possible values of variable y. So, if we "promote" each a type variable to b, and if we detect a definite type error for type variable b, that error is also a definite error for type variable a.

```
-module(evil_erlang).
-export([malicious/0]).

malicious() ->
  file:copy('foo2.beam', 'foo.beam'),
  X = foo:foo(6.0),
  file:copy('foo1.beam', 'foo.beam'),
  X.
```

```
-module(foo). % compiled and renamed to foo1.beam
-export([foo/1]).

foo(X) -> X rem 2.
```

```
-module(foo). % compiled and renamed to foo2.beam
-export([foo/1]).

foo(X) -> X * 7.
```

For Dialyzer to generate a false positive, we just have to compile both `foo` modules, and rename the first one to foo1.beam and the second one to foo2.beam. Dialyzer detects that in the first foo module, the rem operator will be applied to a float, a type error in Erlang. But our code changes the module before invoking the method (in a “dynamic hot code swapping” manner), and does not result in a type error. Similar behaviour might be achieved if we invoke the compiler from within the Erlang runtime system, and the behaviour is in no way different from any “eval magic”, as in both examples we execute code unavailable to the static analysis tool.

Figure 4.1: Confusing the Dialyzer

Unfortunately, this definition leads to extremely counter intuitive subtyping relations, but it has been proven to work well in practice. For example, although it is obvious that $\{1, 2, 3\} \sqsubseteq_S Natural$, subtyping in higher order is counter intuitive. For example, in function hierarchy we have something like this: $\{1, 2, 3\} \rightarrow \{4, 5, 6\} \sqsubseteq_S Natural \rightarrow Natural$. The object hierarchy is far worse: $\{foo : \{1, 2\} \rightarrow \{3, 4\}\} \sqsubseteq_S \{foo : Natural \rightarrow Natural, bar : c \rightarrow d\}$. The success subtyping here is almost the opposite of the classic subtyping!

4.5 Success Typings in Core Ruby

The final thing that remains, before presenting the inference mechanism, is to explain how success typings will fit in Core Ruby. We have to see each class as a set of methods. This way, we can use the success subtyping rule to our advantage. An upper bound to the possible types of an object can be determined by the methods of this object, that a part of code tries to call. For each method call we detect, we must force the callee to implement this method. So, its success typing must contain all classes that have this method, with the same arity. While this is difficult to denote, as we cannot possibly know all the classes that have this method, it is easy to add a subtyping constraint that must hold for our object, in order for the program to pass validation.

Chapter 5

Success Typing Inference for Core Ruby

5.1 Constraint Generation

The problem of inferring the success typings can be solved with constraint based type inference rules. Before showing the rules, the typing relation has to be displayed and analyzed.

$$\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e : t \,\&\, C;\ \Gamma'$$

This relation takes the list of classes, the environment, the types of fields, the object in which this operation takes place (the object of the method currently executing) and the expression being inferred as input, and returns a type, a set of constraints and a new environment. A formal analysis of the symbols exists in Figure 5.1.

So, having defined the typing relation, the inference rules, alongside a brief explanation for each rule, follow. The rules can be seen on Figure 5.2.

5.1.1 Literals

The rules concerning the literals of the language (equation 5.1), should be pretty straightforward. The type of nil is the class with no methods, and the type of self is obtained by finding the type of the current class, which is provided as input to the relation, in the list of classes. These rules generate no constraints, and do not change the environment in any way.

$$\begin{aligned}
t &= \mu \mid \alpha \mid A \mid t_1 \vee t_2 \\
\mu &= \emptyset \mid \mu\ ,\ m \mapsto (t_1 \rightarrow t_2\ when\ C) \\
\Phi &= \emptyset \mid \Phi, @f \mapsto t \\
\Gamma &= \emptyset \mid \Gamma, x \mapsto t \\
\Delta &= \emptyset \mid \Delta, A \mapsto \mu \\
C &= \emptyset \mid C_1 \vee C_2 \mid C_1 \wedge C_2 \mid \{t_1 \sqsubseteq_S t_2\} \mid \{t \neq \{\}\} \mid \{t_1 = t_2\} \mid \{t_1 \in t_2\}
\end{aligned}$$

t : type	μ : list of methods	α : type variable	A : class name
m : method name	Φ : list of fields	$@f$: field name	Γ : list of local variables
Δ : list of classes	C : constraints		

Figure 5.1: Typing relation symbols

$$\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} nil : \{\}\ \&\ \emptyset;\ \Gamma \quad \frac{(l \mapsto m) \in \Delta}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} self : m\ \&\ \emptyset;\ \Gamma} \tag{5.1}$$

$$\frac{(x \mapsto t) \in \Gamma}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} x : t\ \&\ \emptyset;\ \Gamma} \quad \frac{(@f \mapsto t) \in \Phi}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} @f : t\ \&\ \emptyset;\ \Gamma} \tag{5.2}$$

$$\frac{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e : t\ \&\ C;\ \Gamma'}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} x = e : t\ \&\ \emptyset;\ \Gamma'\{x \mapsto t\}} \quad \frac{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e : t\ \&\ C;\ \Gamma' \quad (@f \mapsto s) \in \Phi}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} @f = e : t\ \&\ C \wedge \{t \in s\};\ \Gamma'} \tag{5.3}$$

$$\frac{(A \mapsto m) \in \Delta \quad \Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} m.initialize(e_0, ..., e_n) : tc\ \&\ C;\ \Gamma'}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} A.new(e_0, ..., e_n) : m\ \&\ C;\ \Gamma'} \tag{5.4}$$

$$\frac{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e_1 : t_1\ \&\ C_1;\ \Gamma_1 \quad \Delta;\ \Gamma_1;\ \Phi \underset{l}{\vdash} e_2 : t_2\ \&\ C_2;\ \Gamma_2}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e_1; e_2 : t_2\ \&\ C_1 \wedge C_2;\ \Gamma_2} \tag{5.5}$$

$$\frac{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e_{obj} : t_{obj}\ \&\ C_{obj};\ \Gamma_{-1} \quad \Delta;\ \Gamma_{i-1};\ \Phi \underset{l}{\vdash} e_i : t_i\ \&\ C_i;\ \Gamma_i\ \forall i \in 0, ..., n \quad t_r = fresh()}{\Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e_{obj}.m(e_0, ..., e_n) : t_r\ \&\ C_{obj} \wedge C_0 \wedge ... \wedge C_n \wedge \{\{m : (t_0, ..., t_n) \rightarrow t_r\} \sqsubseteq_S t_{obj}\};\ \Gamma_n} \tag{5.6}$$

$$\frac{\begin{array}{c} t_a = fresh() \quad \Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} e_c : t_c\ \&\ C_c;\ \Gamma_c \quad \Delta;\ \Gamma_c;\ \Phi \underset{l}{\vdash} e_t : t_t\ \&\ C_t;\ \Gamma_t \\ \Delta;\ \Gamma_c;\ \Phi \underset{l}{\vdash} e_f : t_f\ \&\ C_f;\ \Gamma_f \quad \langle \Gamma_n, \langle C_{nt}, C_{nf} \rangle \rangle = \Gamma_t \sqcup \Gamma_f \end{array}}{\begin{array}{c} \Delta;\ \Gamma;\ \Phi \underset{l}{\vdash} if\ e_c\ then\ e_t\ else\ e_f : t_a\ \& \\ C_c \wedge ((\{t_c \neq \{\}, t_a = t_t\} \wedge C_t \wedge C_{nt}) \vee (\{t_c = \{\}, t_a = t_f\} \wedge C_f \wedge C_{nf}));\ \Gamma_n \end{array}} \tag{5.7}$$

$$\frac{\begin{array}{c} \forall x \in dom(\Gamma_t) \cup dom(\Gamma_f) :\ \exists t_n : (x \mapsto t_n) \in \Gamma_n \wedge \\ (\forall t_t : (x \mapsto t_t) \in \Gamma_t \Rightarrow \{t_t = t_n\} \in C_{nt}) \wedge (\forall t_f : (x \mapsto t_f) \in \Gamma_f \Rightarrow \{t_f = t_n\} \in C_{nf}) \end{array}}{\langle \Gamma_n, \langle C_{nt}, C_{nf} \rangle \rangle = \Gamma_t \sqcup \Gamma_f}$$

Figure 5.2: Core Ruby type inference rules

5.1.2 Variables

Next, the rules about the variables (5.2). These are also simple rules, in that they neither change the environment, nor generate any constraint just like the literal rules.

In order to find the type of a local variable, we just have to search our environment, Γ, which contains all the mappings from local variables to types. And, in order to find the type of a field, we search the field environment Φ for the occurence of our field.

5.1.3 Assignments

The assignment rules are more complex (5.3). In both local variable and field assignments, we have the same prerequisite. We must infer the type, constraints and new environment of the right expression.

Then, in local variables, things are fairly easy. We just have to return a new environment, which contains the new mapping, from our local variable to its new type.

In field assignment, though, things are a bit trickier. We cannot just change the field environment. The problem is that there is no guarantee that, later in the inference, a field will not change types. As a matter of fact, due to extreme aliasing in the language, a field type can not be guaranteed to stay the same after any method call, unless a full aliasing analysis is performed. Just for the record, consider the example in Figure 5.3.

So, instead of bookkeeping the current type of each field, the easiest sound solution is to collect all the possible types a field can have, throughout the execution of the program. And in order to do that, our course of action in a field assignment is to simply generate a constraint that denotes that we just found another possible type for the field.

Note that, if the program has a setter, a method that sets a field to its argument, then the type of that field cannot be constrained. A setter can be called from everywhere in the program, and since we cannot have a complete call graph, it is impossible to collect all the types for that field.

But the fields that have setters are not the only possible fields in a class. For example, in a linked list, the field pointing to the next node can usually be determined to be of `List` or `NilClass` type. So, when a method of this field is called, the type of the field can be temporarily assumed as List, as NilClass holds no methods in Core Ruby. Of course, after the method call, there is no way to guarantee that the field has not changed type.

5.1.4 New

The only difficulty found in the rule about `new` (5.4) is that we have to find the constraints of the constructor call. The type of the class is found in the list of classes, the constructor call constraints are generated, and the return type is the type originally found in the list of classes.

Actually, the constructor call constraints will be the constraints produced by the type inference of the arguments, and a subtyping relation, which forces our object to implement the `initialize` method, with the correct arity. Of course, by default, each class should implement a default constructor with no arguments. This is a minor detail, not concerning the theory.

5.1.5 Semicolon

The semicolon rule (5.5) should be no different than a rule from a static type system. The inference of both operands is required, keeping all constraints but discarding the type of the first expression. There

```ruby
class Foo
        def m(u)
                u.set(Bar.new)
        end
end

class Bar
        def m(u)
                u.set(Six.new)
        end
end

class Six
  def *(s)
    FortyTwo.new
  end
end

class Seven
end

class FortyTwo
end

class Foobar
        def foo(u)
                @f = Foo.new
        end

        def bar(u)
                @f = Bar.new
        end

        def chg(u)
                @f.m(u)
        end

        def set(x)
                @f = x
        end

        def check(u)
                @f.*(Seven.new)
        end
end

foo = Foobar.new;

foo.foo(nil);
foo.chg(foo);
foo.chg(foo);
foo.check(nil)
```

It is difficult to impossible to follow the type of the field. Most tools will detect a false positive, while the program executes flawlessly and returns an instance of FortyTwo class.

Figure 5.3: Core Ruby Aliasing

is nothing more to be said here.

5.1.6 Method Call

Probably the most interesting rules are the rules about method calling and branching. In a method call (5.6), it should be made clear that the inferencer can in no way know what code will be called, at the time of constraint generation. That is because, in the general case, the type of the object whose method is called is not known before solving the constraints.

For example, if the programmer calls a method of the argument of a method, it is not possible to know the type of the argument without instantiating the first method call. After all, in the Core Ruby syntax, the callee of a method can be a complex expression, which cannot be resolved while generating the constraints.

The point of all this is that it is not possible to find the correct code to analyze, or the return type of a method, at least not in this phase of type checking. Instead, a more general approach must be followed. By generating new variables and the right constraints, all heavy work can be outsourced to the constraint solver. The solver can perform the correct substitutions, and instantiate the method in order to fully type check the program, if it can reliably determine the type of the callee object.

5.1.7 If-Then-Else

Last, but not least, there is the branching rule (5.7). This is by far the most complex rule, and also the most difficult rule to express in formal language.

The first part should be pretty straightforward. First of all, we infer the type, constraints, and output environment of the condition. Then, we infer both clauses of the statement with the same input environment, which is the output environment of the condition. This should not be very different from the corresponding static typing rule.

Then, there lies the obscure part of merging the two environments, an act which also generates constraints. The idea is simple. When merging the environments of an if-then-else statement, for every variable that exists in at least one of the two environments, produced by the two clauses of the if statement, an entry is added to the output environment, and an equality constraint is added to the respective constraint set that keeps the type it had at the end of the respective clause. It is up to the solver to handle the rest, and combine the two constraint sets.

Finally, the last part of the output constraints should be straightforward. We have a disjunction. We either have a condition that is not nil, while the constraints of the true expression must be valid, as well as the constraint that the output type must be equal to the output type of the true expression. Or, we have a condition that is nil, alongside the obligation that the constraints of the false expression hold, and the output type is the same as the output type of the false condition.

5.2 Constraint Preprocessing

Before solving the constraints, there is a way to simplify the inference. By performing a single pass to the constraints, a preprocessor can lift all the $t_1 \in t_2$ constraints, by calculating the largest set of classes a field can possibly have. This may seem suboptimal, but it guarantees a much easier handling of fields.

The only corner case in this preprocessing is that, when a possible type for a field is the type variable of a method argument, then there is no possible way to constrain that field. The reason is simple, since no guarantees exist that there is no aliasing, there is no possible way to trace back all the possible calls of this particular method. This is a must, if the no false positive policy is to be preserved. Figure 5.3 is a good example about this.

Unfortunately, since no analysis is performed on each individual method before the whole constraint set is processed, at this point, every type variable must be assumed as "tainted", and the field should be assumed to be of the "__AnyClass" class. Of course, this should not prove too difficult to change in future work, and perform an analysis on each individual method, hoping that some type variables will be substituted with known types.

Also, the return type of every function can be substituted with a new variable, and an equality constraint can be generated, in order to handle corner cases where the argument type is the same as the return type, or when the return type is known at generation time.

5.3 Constraint Solving

Solving the constraints is not trivial. The problem is that it is very difficult to express the solver in a formal language, the main culprit being the "when" clause of constraints. Unfortunately, the absence of formal definition of the "when" clause propagates, so it is impossible to prove some implicit assumptions mathematically. Also, it is difficult to determine the order the constraints must be handled. Instead, the solver will be presented intuitively. So, the only thing that remains is to define what the solver will do for each type of constraint in the set. The solver begins with the constraints generated by the expression outside of the classes.

The constraint that a type is not nil is probably the most trivial one. If, after some substitutions, the solver ends up having a class as an operand, there is a type error. Of course, since this constraint is generated only in the false clause of an if statement, this type error just means that the condition cannot possibly be false. But, if the operand of this constraint is not a class, but a type variable, or a union of type variables, this constraint should be reserved for later use.

Then, there is equality. Despite the simple nature of equality, it is difficult to handle it in an algorithm. If either operand is a type variable, things are easy. The solver just substitutes the type variable in the whole constraint set with the other type variable. If either operand is of the "`__AnyClass`" type, the constraint is discarded as always true. If both operands are classes, but not the same class, a type error is generated.

It should be noted that the equality of two different classes is not necessarily a type error, even though one is generated. The error is generated because of the implicit assumption that this should never happen, unless one operand is of the "`__NilClass`" type.

Unfortunately, it is difficult to handle other cases, in general. For example, if another preprocessing or solving scheme is ever used, it is difficult to prove equality of an anonymous class and a named class, since the constraints must perfectly match. Such constraint should never occur with the current implementation. Luckily, the success typing theory protects us in that, should we fail to handle an equality and simply discard it, we will not generate a false positive. Worst case scenario, we will fail to detect an error. Of course, while this assumption might be obvious intuitively, it is difficult to prove with mathematics.

The next constraint class is the success subtyping. Of course, if something like $a \sqsubseteq_S b \land b \sqsubseteq_S a$ is detected, it can be promoted to an equality. Again, there is an implicit assumption that this should never happen. Actually, the only expression that should appear as the first operand should be a list of methods, an anonymous class. So, if the second operand is of the "`__AnyClass`" type, the constraint is discarded as true. If it is a type variable, the solver should wait for a substitution. Else, there are two ways to continue. The mathematically correct way is to append a new set of constraints in our current set, which will be defined in short time. Unfortunately, this will make the generation of backtraces very difficult, since we actually perform a breadth first search. Instead, a depth first search schema will be employed, and the solver will "enter" the solving of the method, solve the constraints, and return some equality constraints to the outer solving, for the return values. It should definitely be noted that there is a problem here. If the solver enters the method, while the argument has not been properly substituted, the solver might fail to find some errors. Of course, it should not generate false positives this way.

So, what set of constraints should the solver process, during such a subtyping constraint? For each method, it should solve the constraints in the "when" constraint, by forcing the argument type and the return type in the right operand *to be equal* with the argument type and the return type of the left operand. This seems utterly wrong at first, so a proper explanation must be given.

The whole problem with the "when" clause is that, there are two obligations an argument must comply to, in order for the call to not fail. It should of course be a success subtype of the formal argument. But, also, it should be a success supertype of the inferred success subtypes of the formal argument. That is, it should be between the formal argument, and the formal argument infimum, known by its constraints. Think of this example:

$$\{add : \{\} \to a\} \sqsubseteq_S \{add : b \to c \; when \; \{\{foo : \{\} \to d \; when \; C\} \sqsubseteq_S b, c = d\}\}$$

What this constraint says might seem difficult to comprehend, but it is not. This constraint represents a method call to `add` of an object, which has an instance method called add, so no type error here. But, inside this method, there exists a call to method foo of the argument, while the return value of the foo method is in fact also the return value of the add method (but this fact will not concern us).

Should we perform the intuitive solving inside the add method, we will end up with the following constraints:

$$\begin{aligned}\{\} &\sqsubseteq_S b\\ a &\sqsubseteq_S e\\ \{foo : d \to e \; when \; C\} &\sqsubseteq_S b\end{aligned}$$

Although correct, they cannot detect the type error that we called our method with nil, yet we call the method foo of the nil class. And, it should be obvious by intuition that the following constraint also holds:

$$\{foo : d \to e \; when \; C\} \sqsubseteq_S \{\}$$

This is the constraint that will lead to the type error, and unfortunately the mathematical rules have to be bent in order to generate it. Since the formal argument type and the formal return value type will always be type variables (partly thanks to the preprocessing), and since the formal argument will never be pinpointed to be of a certain class, should an equality be used, the actual argument type will end up with the desired constraints. Even if the formal argument has a definite known type, for example if an annotation language will be used, the equality will still hold. Unless of course a subtyping annotation is used, when we have to actually copy each argument constraint to force it to hold for the actual argument type, and the equation trick would lead to false positives.

So, by recursively solving the "when" clause of the method, and by returning an equality for the return value, the solver will detect definite errors. Of course, this recursion also means that there must be a stop condition, in order to prevent the tool from entering an infinite loop. The best point to stop the recursion seems to be when the solver tries to recurse into a method already checked, with the same type variable as an argument. Also, in the case of solving a method call that has already been solved for the same argument type, but is not a recursive method call, memoisation could be reliably employed in order to produce correct results for the return type faster.

Last, but not least, there can be a union of types as the right operand of the subtyping constraint. There are three courses of action. Either tractability can be chosen, and the constraint can be ignored completely, or all clauses can be executed, until one comes to a solution, or all clauses can be executed, so not only a type error can be detected, but also the return value can be constrained. There is no right answer here. The prototype implementation chooses the final course of action, but this effectively makes the time to solve the constraints exponential in relation to the length of the program, instead of polynomial. Of course, since ignoring a constraint leads to no false positives, there can be a maximum number of union length, or a maximum number of possible paths, before the solver refuses to execute all paths. Of course, in the worst case, this trick does not lessen the execution time in the slightest.

About the disjunction of constraints, mathematically speaking, all the constraints should be solved twice. This will yield the maximum number of type errors. The problem is that this does not only become exponential in the worst case, but it becomes exponential in every case. If there are n if statements, one below the other, then all constraints will have to be solved 2^n times. This is unacceptable. Let's not forget that this is true, even if the if statements are in different methods, provided that these methods are called sequentially. So, a compromise has to be made. Each clause will be solved once, and the environments will be merged at the end of the if clause. This is where the constraints from merging the environment make sense. They provide the equalities needed in order to create union types, where needed. A type error will be propagated only if all disjunctions fail.

Fortunately, the union of types in a subtyping constraint can be handled the same way as the `if` statements, by generalizing the constraint disjunction. That is, such subtyping constraints can be substituted with equivalent disjunctive constraints.

At first, someone might notice that, while avoiding to solve all constraints twice, we are forced to execute a call twice, if the two `if` clauses return different types. So it seems that nothing is gained. But this is not the case. By solving the call twice at the call level, and not at the `if` level, we make sure that a completely irrelevant `if` statement will never double the solving time.

5.4 Generalizing the tool

5.4.1 Overview

While the idea of combining the success typing theory with a subset of Ruby seems to work, it is still very challenging to generalize the tool to work with the full Ruby specification. There are many benefits from such a generalization. First of all, more testcases can be written. And, importantly, it will be easier to compare our tool to existing implementations.

Since the specification does not seem to be a significant member of the Ruby ecosystem, this will probably fail to be satisfied within a reasonable amount of time. So, the bottom up approach will once again find its use, as we will add some features that make more Ruby code parse successfully. But again, we have to compromise, because it is too difficult to parse all of Ruby.

Why is it so difficult to parse all of Ruby? First of all, Ruby was designed with flexibility in mind, and there exist many different ways to express the same thing, something pretty obvious when looking at the variety of loop constructs, as well as the different ways to express a conditional. A simple example is given below. Each line prints the string "hello", one or more times:

```ruby
if true then puts "hello" end
puts "hello" if true
puts "hello" unless false

3.times { puts 'hello' }
0.upto 3 do puts %{hello} end

for x in [1,2,3] do puts "hello" end
for x in 1..3 do puts "hello" end

x = 0; while x < 3 do puts "hello"; x += 1 end
x = 0; loop do puts "hello"; x += 1; if x >= 3 then break end end
```

Listing 5.1: Variety of Ruby constructs

So, at this point, it would be prudent to use an existing tool to parse a Ruby program, and try to create the corresponding abstract syntax tree of Core Ruby, if such exists. Of course, this means that loops cannot be added in a direct way.

Unfortunately, even if we wanted to add loops on top of the current implementation, we need a fixpoint mechanism in order to infer the success typing, otherwise the inferred types will not be all the possible types for each variable. An example:

```ruby
y = "7"
x = [7]

n.times do
  x = y * 6
  y = 7
end
```

Listing 5.2: Necessity of fixpoint in loops

Depending on the number of times the loop is executed, x will either be an array containing the number 7, the string "777777", or the number 42. So, the success typing of the variable x after the loop should be $Array \vee String \vee Fixnum$, and not a subset of it. As for the y variable, it should be of $String \vee Fixnum$ success typing.

So, there are two questions remaining. What the constructs that will be added will be, and if such a tool exists, that will help us on our quest. Fortunately, there exists a tool called simply Ruby Parser [Ruby], and seems to work well for the cases it was tested against.

5.4.2 Adding Constructs

Before discussing the added constructs, we should make sure that our extended tool performs well, even when it fails to parse an expression. Luckily, we can once again fall back to the success typing definition, and just try to make somewhat sure that a parse that fails can be omitted, without producing false positives. This is moderately easy, if we forget about `eval` and metaprogramming. Simply, if a function fails to parse, it will be substituted with a function that returns "__AnyClass". And if a class fails to parse, it will be substituted with "__AnyClass". The latter will usually happen if we encounter static variable initialisations. If a top level expression fails to parse, then we probably cannot do anything but accept the whole program.

So, for starters, by using Ruby Parser, we get some things for free. We can now parse function calls without parentheses, something that is pretty common in Ruby. That also means, we can parse almost all symbol methods, either in method call notation, or in infix notation. And, after that, there are many alternate constructs that can be parsed for free, as for example an if-then-else expression without the `then` keyword.

Since we still don't have loops, we have to add many features to compensate for that. It seems that the easier addition is to add literals, either string or integer, and to add some special literals, such as regular expressions and string literals inside backticks, denoting shell commands. After all, the only thing that we need is an expression that returns the same type, the type of the literal. Since we treat constructors in a special way, we only need to add an invocation of the `new` method of the corresponding class. Of course, the `false` literal will be substituted with a nil, in order to comply with the current implementation.

In order to go further, we must overcome a serious obstacle. Ruby has a standard library, and we need to have some way to avoid generating an error, if we detect a String or a Fixnum object. Unfortunately, there seems no obvious clean way to do so, and a hack must be devised. The ruby interpreter will again be invoked, so that it will have a clean environment, and then will execute a single query to detect whether a type exists, and implements the method being called. After all, all typing information are accessible to any Ruby program at runtime. Of course, we have no way to constrain the arguments or the return type of the standard library call. While it could be possible to find the types from an external source, it is not certain whether a completely up to date source exists, that contains all standard library objects, and the possible argument and return types. After all, a type checker should be able to disbelieve any invalid type annotation.

Chapter 6

Comparison to Existing Tools

6.1 Existing tools

While there exist many tools for analysing a Ruby program, type checking does not seem to have a prestigious position among those tools. The main reason for this seems to be that Ruby, as well as Python and other scripting languages, encourage test driven development. So, type checking seems rather unnecessary. Also, since Ruby makes rapid prototyping easy, tools come and go, and an exceptional tool may appear, never to be updated again.

The only tool that seems to do static type checking, and is almost up to date, is Diamondback Ruby (DRuby) [DRub]. So, it will be our point of reference, as to the viability of our tool. First of all, an overview will be presented, in order to show the state of both tools, based on the results of some testcases. Then, in the next section, the most interesting invocations of both tools will be presented, and analyzed. Finally, some interesting conclusions will be drawn from the points made at the previous sections.

6.2 Overview of Comparison

Before doing a step by step analysis on the output of both tools, it would be interesting to see a couple of graphs on how both tools work with the set of testcases used to test the other tool. Before showing these graphs, though, it should be clarified that most conclusions drawn from these graphs will be wrong. And for that, there are two simple reasons.

First of all, each testbench is tailored to the respective tool. And it is intuitively obvious that a custom testbench is as creative as the creators of the tool. After all, that is one of the reasons our tool is realised, to compete with the unit testing of programs. Also, it is difficult to write meaningful testcases that you know your program cannot possibly succeed, and in some cases, not worth the effort. Nevertheless, both testbenches will also be tested with the official Ruby implementation, to have a better overview of things.

One could argue that we should try to find real world testcases for our tool. Unfortunately, most programs written in Ruby use every possible obscure feature of the Ruby language, to the extent that it is impossible to convert them to Core Ruby. While it would be possible to start adding features, this thesis aims also at finding a theory that can be applied to other languages, and not at devising Ruby specific hacks.

Another problem is that the DRuby testbench is too Ruby specific, to rewrite in Core Ruby. It tests many possible strange aspects of the Ruby language, to the point that it is in an orthogonal direction to our needs. Most DRuby tests for its static analyzer are small programs that test against a specific Ruby feature, and there seem to be no real world examples.

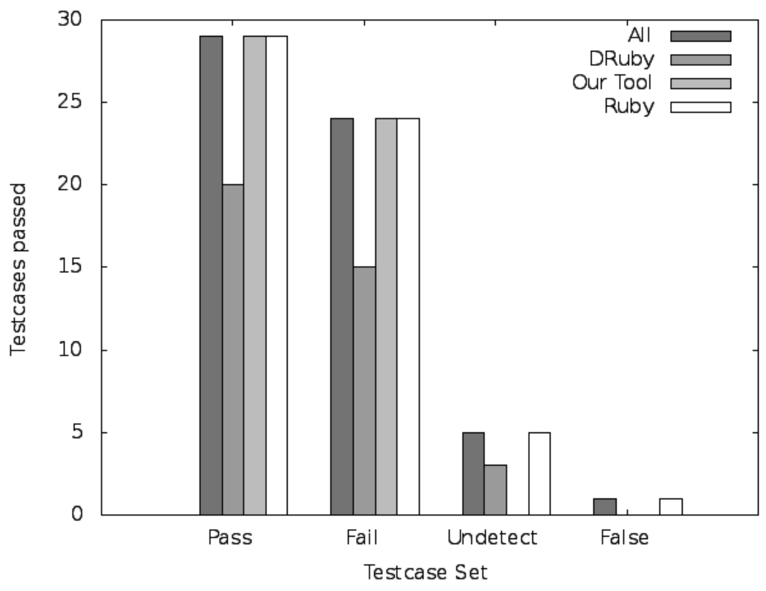

Figure 6.1: Comparison of DRuby with our tool, our testcases (More is better)

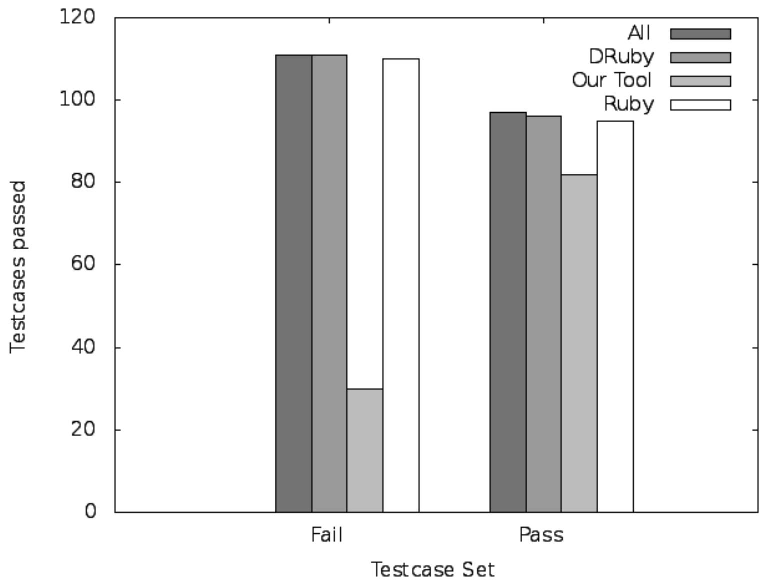

Figure 6.2: Comparison of DRuby with our tool, DRuby testcases (More is better)

Finally, both tools have very different aims, and they compete on different grounds. DRuby aims at finding errors, our tool aims at not finding false positives. Also, DRuby looks at annotations, our tool does not. Most DRuby testcases target metaprogramming, our tool does not handle it at all. The fact is that, after extending our tool with the aid of Ruby Parser, it passed most of the correct DRuby testcases. But this was probably due to its failure to parse the programs, and so it gave up and accepted the program, and not because of some meaningful reason. While this is what the tool is supposed to do, any comparison cannot be considered conclusive, not before we can somehow qualify how big a "give up" is. Also, most of the errors that got caught by our tool are simply wrong, as the error lies someplace else.

After making everything clear with the above disclaimers, the two graphs can be seen in figures 6.1 and 6.2, respectively. Shortly, the testcases of our testbench, in which the two tools gave different results, will be discussed in isolation. The only premature conclusion is that, both tools have something to learn from the other tool.

First of all, we will see the most interesting cases where Diamondback Ruby emits warnings at a correct testcase. Then, we will see all the cases where DRuby fails to find the error. Finally, we will see the testcases that our tool is known to fail to detect, and see how DRuby performs.

6.3 Step by step comparison

6.3.1 Diamondback Ruby false positives

It should be noted, although obvious, that our tool gives no warnings in any program in this section, unless explicitly specified. So, only the DRuby output will be presented. Every testcase in this section runs flawlessly under the ruby interpreter, version 1.9.3p194 (2012-04-20 revision 35410) [x86_64-linux], again unless explicitly specified.

```ruby
class Foo
        def m(u)
                u.set(Bar.new)
        end
end

class Bar
        def m(u)
                u.set(Six.new)
        end
end

class Six
  def *(s)
    FortyTwo.new
  end
end

class Seven
end

class FortyTwo
end

class Foobar
        def foo(u)
                @f = Foo.new
        end

        def bar(u)
                @f = Bar.new
        end

        def chg(u)
                @f.m(u)
        end

        def set(x)
                @f = x
        end

        def check(u)
                @f.*(Seven.new)
        end
end

foo = Foobar.new;

foo.foo(nil);
foo.chg(foo);
foo.chg(foo);
foo.check(nil)
```

Listing 6.1: aliasing.crb

```
[ERROR] instance Foo does not support methods *
  in method call @f.*
  at ./_aliasing.rb:43
  in adding instance method check
  at ./_aliasing.rb:42
  in class definition ::Foobar
  at ./_aliasing.rb:25
  in assignment to @f
  at ./_aliasing.rb:27
  in typing ::Foo.new
  at ./_aliasing.rb:27

DRuby analysis complete.
```

This program has been analyzed numerous times in this thesis. Due to aliasing, all method calls are valid. Diamondback Ruby notifies us that the `Foo` does not support the `*` method. While this is true, the method `*` is never called with class `Foo` as the callee.

```
class Foo
  def f(u)
    nil
  end

  def g(u)
    nil
  end
end

class Bar
end

x = Foo.new;
y = Foo.new;

```

```
17 if y then
18   (x.f(y); x.g(y))
19 else
20   x.g(y)
21 end;
22
23 if y then
24   x = Foo.new
25 else
26   x = Bar.new
27 end;
28
29 x.f(y)
```

Listing 6.2: if.crb

```
[ERROR] instance Bar does not support methods f
  in method call x.f
  at ./_if.rb:29
  in merging environments
  at ./_if.rb:23
  in typing ::Bar.new
  at ./_if.rb:26

DRuby analysis complete.
```

This program clearly demonstrates that DRuby fails to merge the environments of an `if` statement properly. The variable `y` is initialised as `Foo`, and all if statements will execute the true clause. So, `x` will never be of `Bar` type.

```
7.+(7).+(7).+(7).+(7).+(7)
```

Listing 6.3: corelib number 1.crb

```
<expr>: 2
MC(MC(MC(uvar(Nat).lvar(new)()).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)(
)).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)(
)).lvar(x2p1)(
MC(MC(uvar(Nat).lvar(new)(
)).lvar(zero)(
nil)))))))).+(
MC(block(MC(MC(uvar(Nat).lvar(new)()).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)(
)).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)(
)).lvar(x2p1)(
MC(MC(uvar(Nat).lvar(new)(
)).lvar(zero)(
nil))))))))).+(
))))
MC(MC(MC(MC(uvar(Nat).lvar(new)()).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)(
)).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)(
)).lvar(x2p1)(
MC(MC(uvar(Nat).lvar(new)(
)).lvar(zero)(
nil)))))))).+(
MC(MC(uvar(Nat).lvar(new)()).lvar(x2p1)(MC(MC(uvar(Nat).lvar(new)()).
lvar(x2p1)(
MC(MC(uvar(Nat).lvar(new)()).lvar(x2p1)(
MC(MC(uvar(Nat).lvar(new)()).lvar(zero)(
nil)))))))))).+())
Fatal error: exception Failure(”expr”)
```

There is not much to be said here. DRuby seems to fail to parse an expression like `x.+(y)`, even though it is completely valid Ruby. Nevertheless, if we give DRuby a second chance and rewrite all our programs with arithmetic to use infix notation, this testcase passes.

```ruby
class List
  def clear(u)
    @list = nil
  end

  def insert(x)
    elem = Tuple.new;
    elem.setFst(x);
    elem.setSnd(@list);
    @list = elem
  end

  def addRec(x)
    if x then
      x.getFst(nil).+(self.addRec(x.getSnd(nil)))
    else
      0
    end
  end


  def add(u)
    self.addRec(@list)
  end
end

lst = List.new;

lst.insert(1);
lst.insert(2);
lst.insert(3);
lst.insert(4);
lst.insert(5);
lst.insert(6);
lst.insert(6);
lst.insert(5);
lst.insert(4);
lst.insert(3);
lst.insert(2);
lst.insert(1);

lst.add(nil)
```

Listing 6.4: corelib number 2.crb

```
[ERROR] instance Nat does not support methods getFst
  in method call x.getFst
  at ./_corelib_number_2.rb:15
  in adding instance method addRec
  at ./_corelib_number_2.rb:13
  in method call self.addRec
  at ./_corelib_number_2.rb:23
  in adding instance method add
  at ./_corelib_number_2.rb:22
  in class definition ::List
  at ./_corelib_number_2.rb:1
  in method call lst.add
  at ./_corelib_number_2.rb:42
  in typing return statement: return self
  at ./_corelib.rb:25
  in adding instance method x2
  at ./_corelib.rb:22
  in class definition ::Nat
  at ./_corelib.rb:21

DRuby analysis complete.
```

This however does not. According to the backtrace, there is no need to present the Core Ruby core library, since the error is in line 15. It is not easy to see that `addRec` is called with `@list` as the parameter, and `@list` can only be a tuple that consists of an element and a pointer to the next list tuple. Also, it should be noted that the core library does not use the + methods anywhere, so this is not the case of DRuby failing to parse the core library. After all, the core library passes DRuby validation.

```ruby
class A
  def a(u) nil end
end

class B
  def b(u) u end
end

class C
  def c(u) self end
end

class Pair
  def setFst(x) @fst = x end
  def setSnd(y) @snd = y end
  def getFst(u) @fst end
  def getSnd(u) @snd end
end

class Main
  def main(x) x.getSnd(nil).b(x) end
end

p = Pair.new;
p.setFst(A.new);
p.setSnd(B.new);
Main.new.main(p);

p.setFst(C.new);
Main.new.main(p);

p.setSnd(C.new);
Main.new.main(p)    # this must fail!
```

Listing 6.5: tuple.crb

```
[ERROR] instance C does not support methods b
  in method call b
  at ./_tuple.rb:21
  in method call x.getSnd
  at ./_tuple.rb:21
  in adding instance method main
  at ./_tuple.rb:21
  in class definition ::Main
  at ./_tuple.rb:20
  in method call main
  at ./_tuple.rb:27
  in typing ::C.new
  at ./_tuple.rb:32

DRuby analysis complete.
```

And one final testcase. This testcase is from the “undetected” set of Core Ruby, as it crashes, and our tool cannot detect it. The funny thing is that, although DRuby detects an error, the error it detects is on the wrong line. And, even if we comment out the final line, which crashes, DRuby continues to detect an error. So, this testcase is also considered a false positive for the DRuby type checker. This testcase is very intriguing. It shows that Diamondback Ruby determines the type of a field of a class, by analysing what types of fields will not lead to the methods crashing.

This fact inevitably leads to rhetorical questions. While this is a clever approach, it contains the pitfall that detects an error in this testcase. DRuby never checks whether those methods are actually called, after the field is assigned a different type than the set inferred. And, unfortunately, it is not a rare case for a program to have implicit assumptions on the state of an object before a method invocation. Unfortunately, while DRuby seems to enforce good practice, it defeats in the process, in part, the purpose of using a dynamically typed language. After all, dynamic typing is usually about rapid prototyping, and not about creating concrete solutions from the beginning, especially when used in scripting.

6.3.2 Diamondback Ruby false negatives

While there exist many testcases where both tools detect the error, it seems more interesting to see the testcases where our tool detects the error, and not DRuby. In this section, only the output of our tool will be presented, since DRuby prints a plain “DRuby analysis complete.”.

```ruby
class Empty
end

class Foo
  def foo(x)
    x.foo(nil);
    nil
  end
end

x = Foo.new;
y = Foo.new;

# Following call will fail
x.foo(y)
```

Listing 6.6: second order.crb

```
Line 6: Type error detected: class __NilClass has no method foo/1
                at line 6
                at line 15
```

Interestingly, DRuby fails to detect this obvious error. It could probably detect the error if the method had an annotation. The error should be pretty straightforward. Unfortunately, since our tool focuses on this type of error, we have many testcases of this type, and there seems no reason to present them all.

```ruby
class Foo
```

```
  def foo(x)
    x.init(nil)
  end
end

x = Foo.new;

# Following line should fail because NilClass has no init method
x.foo(nil)
```

Listing 6.7: method.crb

```
Line 3: Type error detected: class __NilClass has no method init/1
                at line 10
```

This seems to be the most simple case of the previous behaviour of DRuby. It won't be analyzed any further.

```
x = nil;

if x then
  nil
else
  x.foo(nil)
end
```

Listing 6.8: full deadcode.crb

```
Line 7: A disjunction failed:
        True:   Condition is always false
        False:  Line 6: Type error detected: class __NilClass has no method foo/1
        End
```

DRuby fails to detect the error. Maybe it does not consider `nil` to trigger a false condition. It is difficult to analyze with no prior knowledge of the DRuby code base.

```
class Foo
  def f(u)
    u.f(u)
  end
end

class Bar
  def f(u)
    u.f(nil)
  end
end

class FooBar
  def do(u)
    y = if @x then
      Foo.new
    else
      Bar.new
    end;
    y.f(Bar.new)
  end

  def set(x)
```

```
24     @x = x
25   end
26 end
27
28 FooBar.new.do(nil)
```

Listing 6.9: full deadcode 4.crb

```
Line 20: A disjunction failed:
        Foo:    Line 9: Type error detected: class __NilClass has no method f/1
                        at line 9
                        at line 3
                        at line 20
        Bar:    Line 9: Type error detected: class __NilClass has no method f/1
                        at line 9
                        at line 20
        End
                at line 28
```

While the output seems funny, the case is that our tool cannot detect for sure whether the true clause of the `if` statement will be executed, or the false clause. The reason is that we have a setter, so the field type cannot be pinpointed. But both clauses lead to crashes, so our tool says exactly that.

```
class Tree
  def addLeft(left)
    @left = left
  end

  def addRight(right)
    @right = right
  end

  def setData(data)
    @data = data;
    self
  end

  def addRec(u)
    sum = sum.+(@data);
    if @left then sum = sum.+(@left.addRec(nil)) else nil end;
    if @right then sum = sum.+(@right.addRec(nil)) else nil end;
    sum
  end
end

a0 = Tree.new.setData(0);
a1 = Tree.new.setData(1);
a2 = Tree.new.setData(2);
a3 = Tree.new.setData(3);
a4 = Tree.new.setData(4);
a5 = Tree.new.setData(5);
a6 = Tree.new.setData(6);

a0.addLeft(a1);
a0.addRight(a1);

a1.addLeft(a2);
a1.addRight(a3);

a2.addLeft(a4);
a2.addRight(a5);
```

```

a3.addLeft(a6);

a0.addRec(nil)
```

Listing 6.10: tree undef.crb

```
Line 16: Variable sum does not exist
```

Again, a simple case that DRuby fails to detect. This is the tree example already presented, with the initialisation of the `sum` variable missing. The actual error of Ruby would be that NilClass does not implement the + method, since every fresh r-value is initialised as nil. Nevertheless, this is still an error, no matter the details of the language semantics.

This testcase concludes the presentation of the false negatives of DRuby.

6.3.3 Our tool false negatives

```
class List
  def tabulate(tup) # { funobj, count }
    @data = tup.getFst(nil).fun(tup.getSnd(nil));
    if tup.getSnd(nil).iszero(nil) then
      @next = nil
    else
      @next = List.new;
      tup.setFst(tup.getFst(nil).-(1));
      @next.tabulate(tup)
    end;
    self
  end

  def foldl(tup) # { funobj, acc }
    if @next then
      tup.getSnd(nil)
    else
      accelem = Tuple.new;
      accelem.setFst(@data);
      accelem.setSnd(tup.getSnd(nil));

      tup.setSnd(tup.getFst(nil).fun(tup));

      @next.foldr(tup)
    end
  end
end

class Double
  def fun(x)
    x.+(x)
  end
end

class Sum
  def fun(tup)
    tup.getSnd(nil).+(tup.getFst(nil))
  end
end

tup = Tuple.new;
```

```
tup.setFst(6);
tup.setSnd(Double.new);
lst = List.new.tabulate(tup);

tup = Tuple.new;
tup.setFst(Sum.new);
tup2 = Tuple.new;
tup2.setFst(lst);
tup2.setSnd(0);

tup.setSnd(tup2);

lst.foldl(tup)
```

Listing 6.11: undetected/corelib higher list.crb

```
[ERROR] instance Nat does not support methods fun
  in method call fun
  at ./_corelib_higher_list.rb:3
  in method call tup.getFst
  at ./_corelib_higher_list.rb:3
  in adding instance method tabulate
  at ./_corelib_higher_list.rb:2
  in class definition ::List
  at ./_corelib_higher_list.rb:1
  in method call tabulate
  at ./_corelib_higher_list.rb:44
  in typing return statement: return self
  at ./_corelib.rb:25
  in adding instance method x2
  at ./_corelib.rb:22
  in class definition ::Nat
  at ./_corelib.rb:21

DRuby analysis complete.
```

There are many things wrong with this testcase. First of all, the `tabulate` method, in line 8, has the tuple arguments messed up. It should have invocations of the `setSnd` and `getSnd` methods, instead of `setFst` and `getFst`. Also, in the foldl method, the clauses of the if statements are reverse. Should the false clause execute, the `@next` field would be nil, and an exception would be raised. Unfortunately, there is no way for our tool to tell whether the false clause will be executed at all, and no warning is emitted.

On the other hand, DRuby seems to catch the first error. The backtrace does not help much, though.

```
class List
  def clear(u)
    @list = nil
  end

  def insert(x)
    elem = Tuple.new;
    elem.setFst(x);
    elem.setSnd(@list);
    @list = elem
  end

  def addRec(x)
    if x then
      x.getFst(nil).+(self.addRec(x.getSnd(nil)))
    else
```

```
      0
    end
  end


  def add(u)
    self.addRec(@list)
  end
end

lst = List.new;

lst.insert(1);
lst.insert(2);
lst.insert(3);
lst.insert(4);
lst.insert(5);
lst.insert(List.new);
lst.insert(6);
lst.insert(5);
lst.insert(4);
lst.insert(3);
lst.insert(2);
lst.insert(1);

lst.add(nil)
```

Listing 6.12: undetected/corelib number 2.crb

```
[ERROR] instance List does not support methods getBit
  in method call snd.getBit
  at ./_corelib.rb:57
  in method call tup.getFst
  at ./_corelib.rb:52
  in method call self.addBit
  at ./_corelib.rb:114
  in adding instance method addRec
  at ./_corelib.rb:103
  in typing return statement: return tup2
  at ./_corelib.rb:99
  in adding instance method addBit
  at ./_corelib.rb:48
  in class definition ::Nat
  at ./_corelib.rb:21
  in typing ::List.new
  at ./_corelib_number_1.rb:34

[ERROR] instance Nat does not support methods getFst
  in method call x.getFst
  at ./_corelib_number_1.rb:15
  in adding instance method addRec
  at ./_corelib_number_1.rb:13
  in method call self.addRec
  at ./_corelib_number_1.rb:23
  in adding instance method add
  at ./_corelib_number_1.rb:22
  in class definition ::List
  at ./_corelib_number_1.rb:1
  in assignment to @list
  at ./_corelib_number_1.rb:10
  in merging environments
  at ./_corelib.rb:56
  in merging environments
  at ./_corelib.rb:57
  in adding instance method addBit
  at ./_corelib.rb:48
  in class definition ::Nat
  at ./_corelib.rb:21

DRuby analysis complete.
```

Here, the problem is more obvious. At line 34, we insert a non-integer element in our list, and the consequences will be definitely fatal for the execution, as the `List` class has no definition for the `+` method. Our tool cannot detect anything, as it cannot constrain the contents of a Tuple class. DRuby seems to detect a discrepancy in line 34 of our program, but the backtrace goes way deep in the Core Ruby library. Unfortunate though it seems, it is pretty acceptable. Our codes have no type annotations, so we cannot have DRuby check each individual module in isolation.

But, DRuby seems to detect an error at line 15. It says that we try to invoke the `getFst` method of a `Nat`. This is not the case, as has been already stated in the collection of false positives of DRuby, in listing 6.4

6.3.4 Interesting cases

Here, we will present some interesting cases that do not fall in either previous category.

```
class Number
  def to_s
    num = 0
    num += if @n7 then 1 else 0 end
    num <<= 1
    num += if @n6 then 1 else 0 end
    num <<= 1
    num += if @n5 then 1 else 0 end
    num <<= 1
    num += if @n4 then 1 else 0 end
    num <<= 1
    num += if @n3 then 1 else 0 end
    num <<= 1
    num += if @n2 then 1 else 0 end
    num <<= 1
    num += if @n1 then 1 else 0 end
    num <<= 1
    num += if @n0 then 1 else 0 end
    num.to_s
  end

  def inspect
    self.to_s
  end
end

nil
```

Listing 6.13: numlib.rb

This code snippet is from an early version of the Core Ruby implementation. The strange thing is that, in this testcase, DRuby needs 3 and a half minutes before it concludes that this program does not have any error. There seems to be no obvious reason for that, and there is no way to know what exactly happens without knowledge of the DRuby implementation.

```ruby
class Foo
end

def genm x, y
  eval ”class Foo; def #{x}; #{y}; end; end”
end

x = Foo.new

genm ”six”, 6
genm ”seven”, 7

x.six * x.seven # prints 42
```

Listing 6.14: eval metaprogram.rb

```
Line -1: Type error detected: class Foo has no method six/0
Line -1: Type error detected: class Foo has no method seven/0
```

```
[ERROR] instance Foo does not support methods six
  in method call x.six
  at ../Testcases/Ruby/false_pos/eval_metaprogram.rb:13
  in typing ::Foo.new
  at ../Testcases/Ruby/false_pos/eval_metaprogram.rb:8

[ERROR] instance Foo does not support methods seven
  in method call x.seven
  at ../Testcases/Ruby/false_pos/eval_metaprogram.rb:13
  in typing ::Foo.new
  at ../Testcases/Ruby/false_pos/eval_metaprogram.rb:8

DRuby analysis complete.
```

This testcase is just presented for completeness. Note that our tool fails to print the line of the error. This is an unfortunate side effect from using Ruby Parser. It should not be too difficult to fix, and is a minor detail.

6.4 Conclusions

So, finally, we can draw conclusions from the previous behaviors of the two tools. Before discussing the shortcomings of each tool, it seems more appropriate to discuss the design choices each team followed. Then, the drawbacks of each approach will be analyzed.

6.4.1 Diamondback Ruby

Design decisions

While both tools were designed for error detection, there is a big difference in the approaches. For instance, DRuby seems to facilitate static analysis as complementary to dynamic analysis, and not as a first class citizen.

Also, it uses static checking in order to perform optimizations, as it might make some dynamic type checks unnessecary. In order to lift the run time checks, DRuby has implemented Ruby on its own.

So, it has full control of the run time environment of each program, should someone choose to use its implementation of Ruby.

DRuby seems to have decided that it should first be able to analyze the whole language successfully, and then try to find many errors automatically. This conclusion is drawn from the fact that most test-cases are focused on specific Ruby features. This choice seems reasonable, considering that DRuby understands a type annotation language, so it leaves the method typing documentation to the programmer.

Finally, DRuby tries to impose a strict type system to Ruby.

Drawbacks

There are a couple of drawbacks of Diamondback Ruby, and usually the reason behind those seems to be one or more of the design decisions.

First of all, Diamondback Ruby performs suboptimally on Core Ruby. While the definitions of what is the core of the Ruby programming language might vary, Core Ruby is a language that models the Ruby object orientation almost to the point. And the fact is that, while DRuby was expected to have only false positives, and detect all the errors in the failing testcase set of our tool's suite, this is not the case. Many cases considered trivial by our tool were difficult for DRuby.

The reason behind this seems to be the design decision of first parsing the whole Ruby specification, and then adding automatic error detection. Core Ruby is definitely way beyond a language that has every difficult to parse Ruby feature, so it probably is orthogonal to the ideal language that DRuby performs optimally.

Secondly, as we already saw in a testcase, DRuby makes very optimistic assumptions on the types of fields. This way, DRuby aims to find errors, considering a type of field that its methods cannot handle to be a hack. As already stated, this is a poor choice for a dynamically typed language that favors rapid prototyping, as in dynamically typed languages, a hack can be much easier than refactoring the code, sometimes without sacrificing in readability. After all, a programmer that chooses a dynamic language, knows what he is getting into.

Then, DRuby fails to parse some valid Ruby expressions, such as `x.+(y)`. As DRuby chose to implement Ruby on its own, this has effectively made Ruby less compliant to the Ruby specification, and, more importantly, less resilient to changes to the Ruby specification.

Finally, since DRuby is actually another Ruby implementations, it carries the drawbacks from such a decision. This decision effectively makes the tool hard to be adopted, as a programmer would be hesitant to use an implementation that will probably not be developed with the same pace as the official implementation, just to be able to have a couple of optimizations. Also, this means that those choosing to use the DRuby implementation are locked in it, and if DRuby suddenly disappears, they will have to port their code to the official Ruby implementation, which will probably have changed a lot by then. This is a heavy weight to bear for a simple analysis tool.

6.4.2 Our tool

Design decisions

Our tool, on the other hand, was designed to be an easy to use tool that detects only definite errors. This way, it can be used by anyone as a means of finding some bugs. Also, its shortcomings can be hidden by just accepting what it cannot understand.

Also, it was designed to be fast. At this point, our tool is way faster than DRuby. Of course, DRuby is a much more complete program, while our tool is at its infancy, so there is no guarantee that this will be the case later.

Finally, it was designed to be a static only tool. This was not due to some misconception that Ruby can be efficiently analyzed by employing a static only solution. It was simply a choice, as Ruby is used most of the time on small scripts, in which it would be an overkill to write unit tests. After all, it would be an overkill to unit test a bash script. And it was an experiment, whether a static only solution would be viable.

Drawbacks

Of course, we cannot deny that our decisions have inevitably led to drawbacks. So, we will discuss them further right now.

Our tool detects only definite errors, and ignores what it cannot understand, Also, when `if` clauses or fields are concerned, it currently will fail to detect even very obvious errors. So, it is inevitable that someone will classify it as a "toy tool" right away, as there is no guarantee as to which errors it will find, if it finds any.

Secondly, our tool does not work with full Ruby. Even with the principle that it accepts all strange programs, it detects errors where it should not. This makes it effectively useless, and also difficult to check whether it works, as it will fail on most real world examples. This is a direct result of our bottom up approach to the challenge.

Finally, our tool has probably no hope of making a useful optimizer. The theory behind it, the success typings, make no guarantees about the type safety of a program. So, even if there was an easy way to alleviate the dynamic type checks from the Ruby runtime, our tool would not be able to use it efficiently.

Chapter 7

Conclusion

7.1 Contribution

So, to conclude, while this comparison seems to remain inconclusive until our tool parses the full extent of Ruby, it does not. It actually shows that both tools have something to learn from the other tool. Also, it shows that the creation of a tool that performs a success typing inference on Ruby, and finds a bunch of definite errors, is actually feasible.

Also, since our theory works, it could easily be adapted to any dynamically typed object oriented language, or even any dynamically typed language with higher order constructs.

7.2 Future Work

There is much work that can be done from now on. We will name a few in the following paragraphs.

First of all, all Core Ruby support should be dropped. The tool currently has two parsers and one runtime. The Core Ruby parser and runtime should be dropped in favor of more clean codebase. This way, it will be much easier to support a bigger subset of the Ruby language. Or, maybe, try to find another language that is less cluttered with features.

Secondly, our tool should try to cope better with the Ruby standard library. In high priority lies the `require` method, and its family. For this method, we will probably need to perform some hacks. For the rest of the standard library, we could try to utilize better the existing solutions, namely, DRuby annotations, as standard library methods are mostly statically typed. In order to work with the DRuby annotations, we just have to check them once, in order to avoid giving bad errors, and then take them for granted.

Then, there are many ways that the solving process can be optimized. First of all, at this point, the solver performs its analysis by beginning from the top level expression, and then instantiating each method when called. While this will detect most errors, it cannot detect dead code, since it cannot prove that an if statement can only be solved one way, regardless of the argument. So, our tool must first construct a call graph approximation, and then try to solve the methods in the right order, optimizing the constraints in the process, for the final analysis. Also, this way, it might be possible to better finegrain the possible field types.

Also, there are many minor changes that can be made, so that the tool will be much more user friendly. A simple example is to include the types of the arguments when presenting the programmer with a stack trace. Another example is to try to print the code part that is in error.

Finally, it is only logical that we should try to give back to the tool we got our inspiration, namely the Dialyzer. The Dialyzer does not cope well with higher order functions, as those are not used so extensively as in Ruby. Maybe our theory can be applied back to functional languages.

Bibliography

[An11] David An, Avik Chaudhuri, Jeffrey Foster and Michael Hicks, "Dynamic Inference of Static Types for Ruby", in *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pp. 459–472, ACM, 2011.

[Dial] "The Dialyzer, a DIscrepancy AnaLYZer for ERlang programs", `http://www.erlang.org/doc/man/dialyzer.html`.

[DRub] "Diamondback Ruby", `http://www.cs.umd.edu/projects/PL/druby/`.

[Furr09] Michael Furr and et al., "Static Type Inference for Ruby", 2009.

[Lind06] Tobias Lindahl and Konstantinos Sagonas, "Practical type inference based on success typings", in *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pp. 167–178, New York, NY, USA, 2006, ACM Press.

[Ruby] "Ruby Parser", `https://rubyforge.org/projects/parsetree/`.